



Barry McManus

USING DEFENSIVE APPROACHES TO BUILD SECURITY INTO COMPUTERISED SYSTEMS: AUDITOR TIPS

Computerised System IT operations seek to assure data integrity by concentrating time and effort on the security of the network components. This paper seeks to shift some of that focus onto the software component of computerised systems and identify tasks that an auditor can check for.

Data needs to be kept safe from a range of threats: users accidentally enter erroneous data; under pressure staff seek to change data to fabricate results; cybercriminals seek Personal Identifiable Information (PII) to steal patient identities, purchase medicines to sell onto others and to commit fraud; and hackers seek data to disclose secrets. All of these threats exploit vulnerabilities latent within computerised systems and in the various activities used to operate and manage them.

The need for trustworthy and reliable computerised systems is well versed by the regulations and guidance attention on integrity and security^{1,2,3,4,5}.

Organisations define policies, procedures and technical controls to ensure integrity, confidentiality and availability of data and computerised systems. Examples include software monitoring, redundancy, data, encryption, access control, training and administration procedures⁷. The implementing of IT security strategies is big business; Gartner forecasted global spending on IT security and risk management to be approximately \$150 billion in 2021⁸. See Figure 1.

Cyberattacks arguably pose the biggest threat to data integrity and are a daily occurrence. As of 1st February 2022, there were 2030 new Common Vulnerability and Exposures (CVE) received by the US based National Institute for Standards and Technology (NIST)⁹. CVE's are defined as 'weaknesses in the computational logic (e.g. code) that, when exploited, result in a negative impact to confidentiality, integrity, or availability'¹⁰. Logic weaknesses are exploited via cyberattacks.

Cyberattacks are non-discriminate, affecting technologies across business verticals including pharmaceuticals. Examples include:

- (2013) Doctor disabled wifi in former US Vice President's pacemaker to reduce the risk of hacking¹¹
- Ransomware attacks accounted for 50% of all reported security breaches by healthcare organisations between October 2015 and September 2016¹²
- June 2017, Merck was targeted by a ransomware hack ('NotPetya') that shut down computer systems, which led to concerns in the US government¹³
- August 2019, biometric company Suprema, (used by banks and UK police) left one million users' fingerprints, facial recognition and passwords of users on an unencrypted database¹⁴

- January 2020, medical testing company Lifelabs data breach of an estimated 15 million users¹⁵
- August 2020 data centre Equinix data was hit by Netwalker Ransomware with a demand of \$4.5M to prevent the release of compromised data¹⁶
- Feb 2022, Dr Reddy's Laboratories had 'isolated' its data centre services after a cyberattack forced the firm to temporarily shut down operations at its major manufacturing facilities across the world¹⁶.

The increase in data issues is reflected in the attentions of the regulators. The WHO have reported that the number of data related observations have been increasing during inspections¹⁷. Data related observations are evident in FDA 483 Warning Letters, for example: 'failed to exercise appropriate controls over computer or related systems to assure that only authorised personnel institute changes in master production and control records, or other records'¹⁸; 'computer... was not secured such that data files could be deleted without the knowledge of your quality unit'¹⁹; partially testing that network ports would not open with an unauthorised interface²⁰.

FIGURE 1. AVERAGE TOTAL COST OF A DATA BREACH BY INDUSTRY

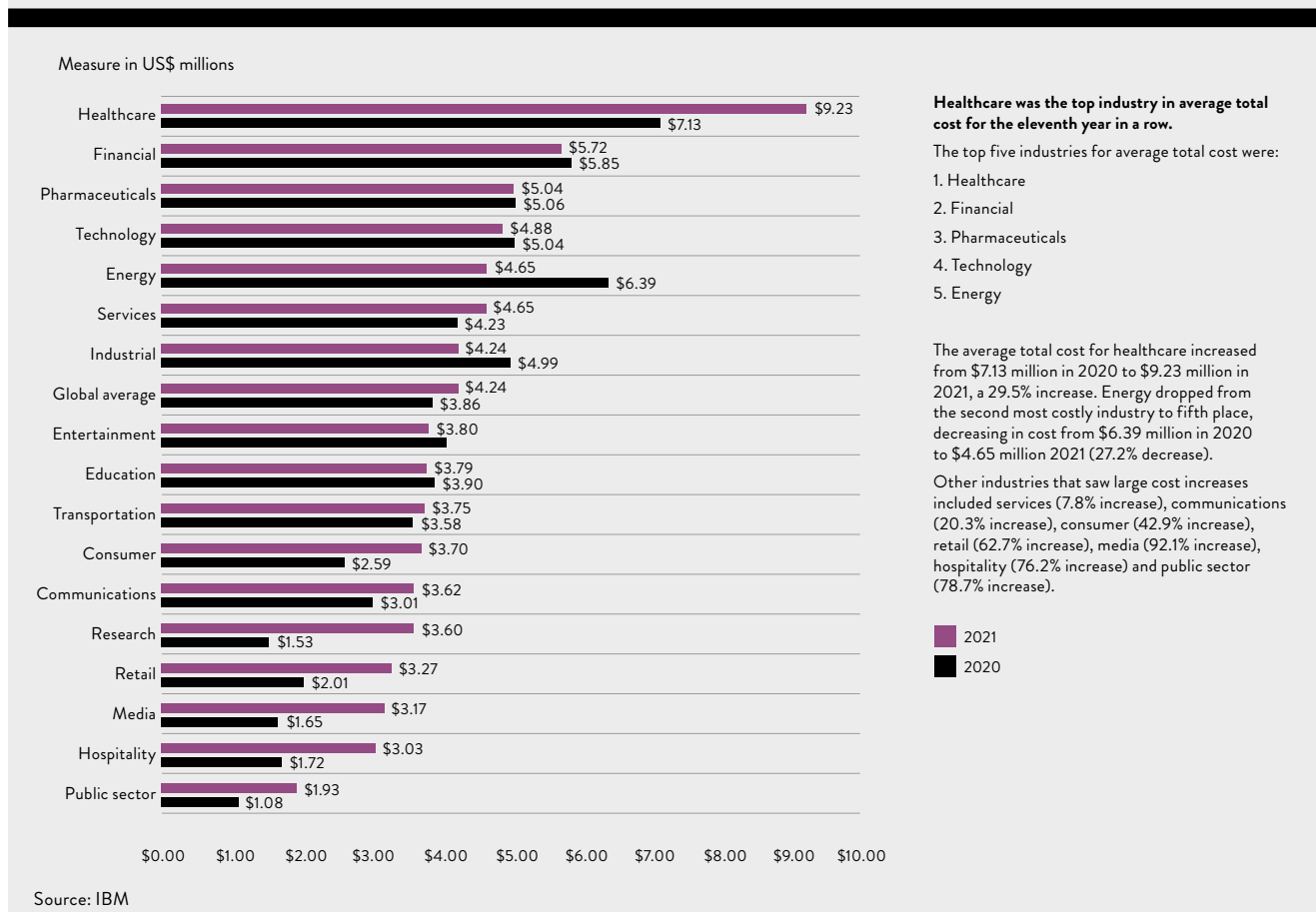
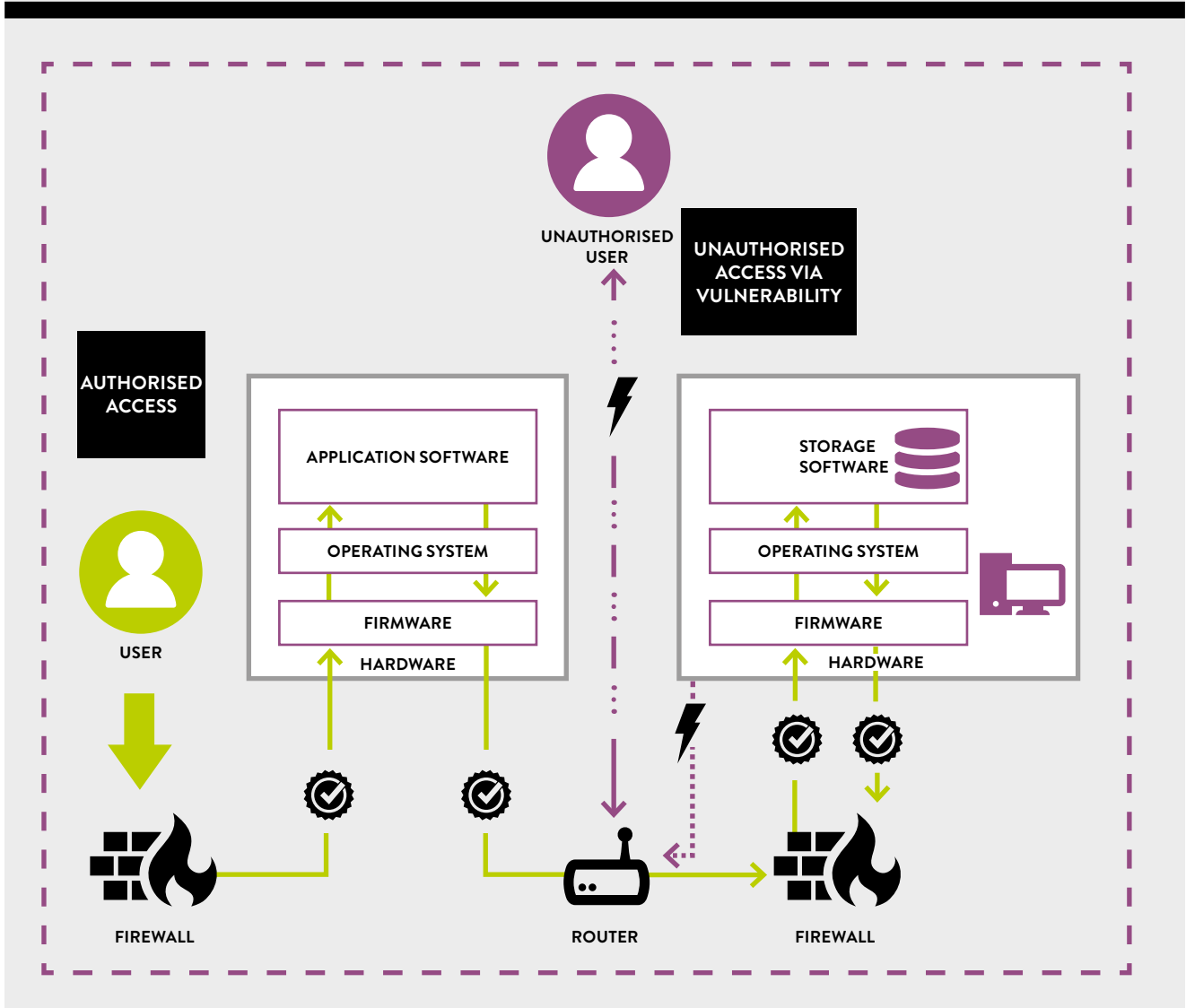


FIGURE 2. EXAMPLES OF EXPLOITING NETWORKED VULNERABILITIES



Given the expenditure on IT frameworks to ‘harden’ computerised systems, coupled with the regulatory emphasis on integrity, confidentiality and availability, why are cybersecurity issues expected to rise in 2022²¹? The answer resides in **software**.

Research opinion is not kind to software: 75% of security breaches occur at the software layer²²; over 92% of reported vulnerabilities exist in software and not in networks²³. Despite the increased spending on network defences, the attacks continue. Outer network defences are ‘cracked’ to access the ‘soft’ inner software (code), for example: to change clinical trial data; switch off audit trails; deny access to records for exploitation; access to PII, for identity theft (for example, birth certificates, credit cards and social security cards cost \$1 on the dark web^{24,25}).

One reason for vulnerable software may be due to computerised systems being used in a way that was never considered. For example, connecting a simple temperature probe with limited code to the network as part of a smart hospital

bed system – a weakness in one connected node of the system can easily compromise other parts of a system (53% of 10 million connected devices within 300 hospitals contained a known vulnerability²⁶).

Examples of exploiting networked vulnerabilities include: a 11-year old boy accessed conference attendees mobile phone contact information using a bluetooth enabled Teddy Bear and a \$35 Raspberry PI computer²⁷; a robust Casino IT system was compromised by exploiting a vulnerability of a networked fish tank²⁸! See Figure 2.

Additionally, vulnerabilities may not be adequately considered during the building of computerised systems. Much of the ‘traditional’ tasks involved in building software have been hidden in 3rd party code libraries which are ‘copied and pasted’, with developers not considering their integrity²⁹. The recent, global impact of the java ‘log4j’ logging library security flaw is a serious manifestation of this practice³⁰.

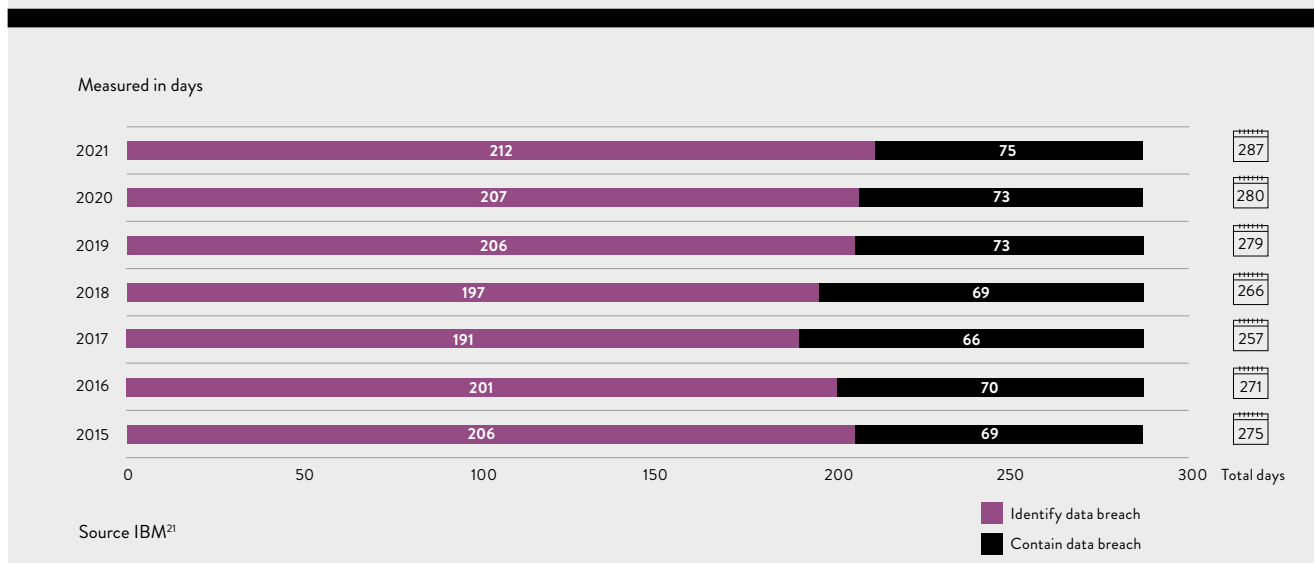
Given that the average length of time that cyberattacks went undetected before containment in 2021 was 287 days³¹, the approach to **building software** demands an alternative look. See Figure 3.

A CHANGE OF MINDSET: SAY HELLO TO MY ‘CRITICAL FRIEND’

Typically, people who write software focus on solutions and how things should work. This positive outlook is necessary, but more is required when seeking to reduce latent vulnerabilities in software that can be exploited.

The use of **Critical Thinking** is a cornerstone for building hardened software. Critical Thinking is to adopt an attitude that is open minded, sceptical and seeks to question all underlying assumptions³².

FIGURE 3. AVERAGE TIME TO IDENTIFY AND CONTAIN A DATA BREACH



For software, this involves expecting anomalies (and seeking to actively mitigate them), ‘thinking like an attacker who is trying to break the application’³³, seeking to understand varying perspectives, challenging the status quo and identifying alternative approaches to problems²⁷. This is a **defensive mindset**.

For computerised systems, the Software Life Cycle (SLC) is the first line of defence as it denotes a framework of human and technical processes to plan, prioritise, design, develop, verify, deploy, accept, maintain and retire computerised systems.

The defence practitioner understands that a gap or lack of rigor in one SLC process will detrimentally impact the success of subsequent activities. Efficient verification is key: inspections and reviews are simple concepts, but when performed correctly, they are the most efficient at removing defects in software by exceeding 60% efficiency in defect removals³⁴ – that is detecting and removing built-in vulnerabilities before production use. Software defects and vulnerabilities are inextricably linked.

A software vulnerability is a defect in software that could allow an attacker to gain control of a computerised system³⁵. Software is prevalent in applications and network components.

The defensive approach focuses on how the data must be complete, consistent and accurate³⁶ by placing emphasis on the source or injection of issue within the following key SLC areas:

- **Quality Approach** – the injection of data problems arise from: poor project management, insufficient requirement reviews, inadequate testing and insufficient version control processes
- **Functional Requirements** – the injection of data problems are evident in: design assumptions, software defects, implementation language deficiencies and operational misuse
- **Non-Functional Requirements** – origins of data vulnerabilities materialise in: electro-magnetic faults, material fatigue, power outages, storage retrieval, performance bottlenecks or malicious system access.

The defensive approach seeks to prevent defects from being designed and built in to the computerised system and it starts with requirements. The defensive mindset seeks to prevent defects in order to prevent vulnerabilities. See Figure 4.

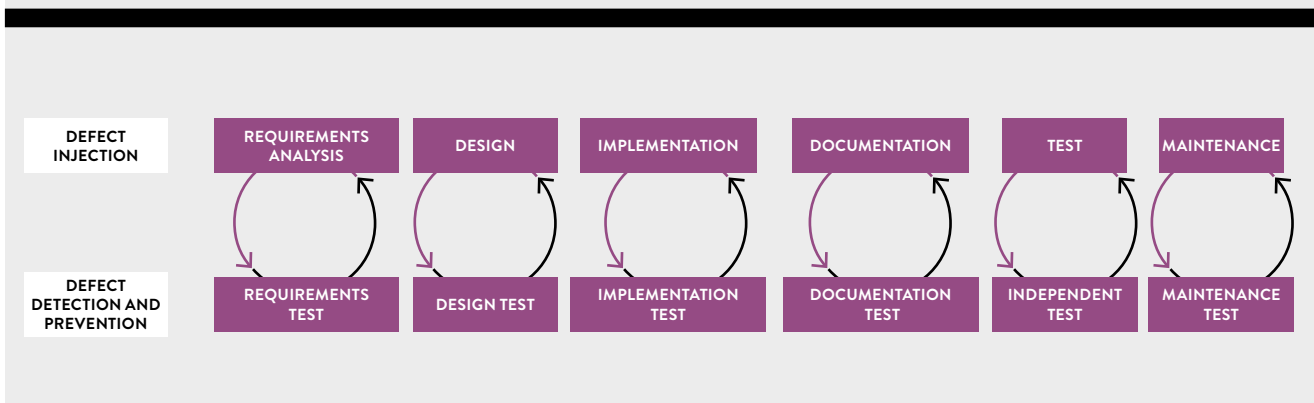
REQUIREMENTS ANALYSIS

Requirement elicitation is the process of studying user needs to arrive at a system definition and quantifying the needs of a system. The defensive mentality seeks to prevent defects at the requirements stage, where it is cheaper to correct and where it reduces the scope of false assumptions propagating multiple defects into software³⁴.

Checklists³⁷ are useful aides to help focus on requirements completeness. For example, when faced with data storage, consider questions such as:

- How will the data be stored (non-functionality)?
- What will the size of the data store be?
- What will happen during processing if the data store size limit is reached?

FIGURE 4. SOFTWARE LIFE CYCLE DEFECT PREVENTION



- What happens if the data store connection is broken?
- How will the user or IT administrators know?
- How will the data in transit not become lost or corrupt?
- Where will the encryption keys be stored and who will access them?
- When were the access logs last reviewed?

Rudyard **Kipling's 5 Ws**: What, Who, Why, When, Where and How³⁸ is an approach to apply to each requirement during a requirements inspection that will help prevent defects from being designed and built into solution. Using the 5 Ws from a negative view point (e.g. what would happen if the computerised system's data store is not available to use) can help focus on reducing the scope of data integrity and security issues. This is a powerful tool for eliciting risks and associated technical control mitigations.

Each requirement should undergo robust risk analysis and this should be an iterative and incremental process throughout the SLC^{37,39}. An example approach is summarised as follows:

1. Assess the business process and associated data according to risk:
 - a. Consider risks and vulnerabilities present in each requirement
 - b. Consider risk and vulnerabilities between components of the business process or proposed system
 - c. Consider the business consequence of data unavailability or leakage
 - d. Consider the probability of risks materialising as issues.
2. As much as possible, devise technical and/or procedural mitigation controls to reduce or eliminate the risks.
3. Establish the tasks to verify that the mitigation controls are implemented: e.g. design reviews, source code reviews, unit testing, creation of automated test harnesses to verify component interfaces, security testing of encrypted traffic between components, stress testing of system memory and so forth.
4. Quantify the cost of the mitigation controls in terms of time, effort and business criticality in order to secure funding for verification activities.
5. Review the requirements, risks and mitigations and concatenate activities together where feasible, such as test scenarios.
6. Seek approval of the costs for the mitigation controls and verification activities with management.

The defensive approach embraces designing user acceptance test scenarios during requirements risk analysis. The task of designing a test is a critical thinking activity that can generate additional questions on the requirement feasibility, reduce the risk of false assumptions, identify gaps within requirements and conflicts between requirements. This simple activity reduces the scope of defects that are designed and built into a computerised system, thus reducing the scope of latent vulnerabilities for possible exploit.

DESIGN STAGE

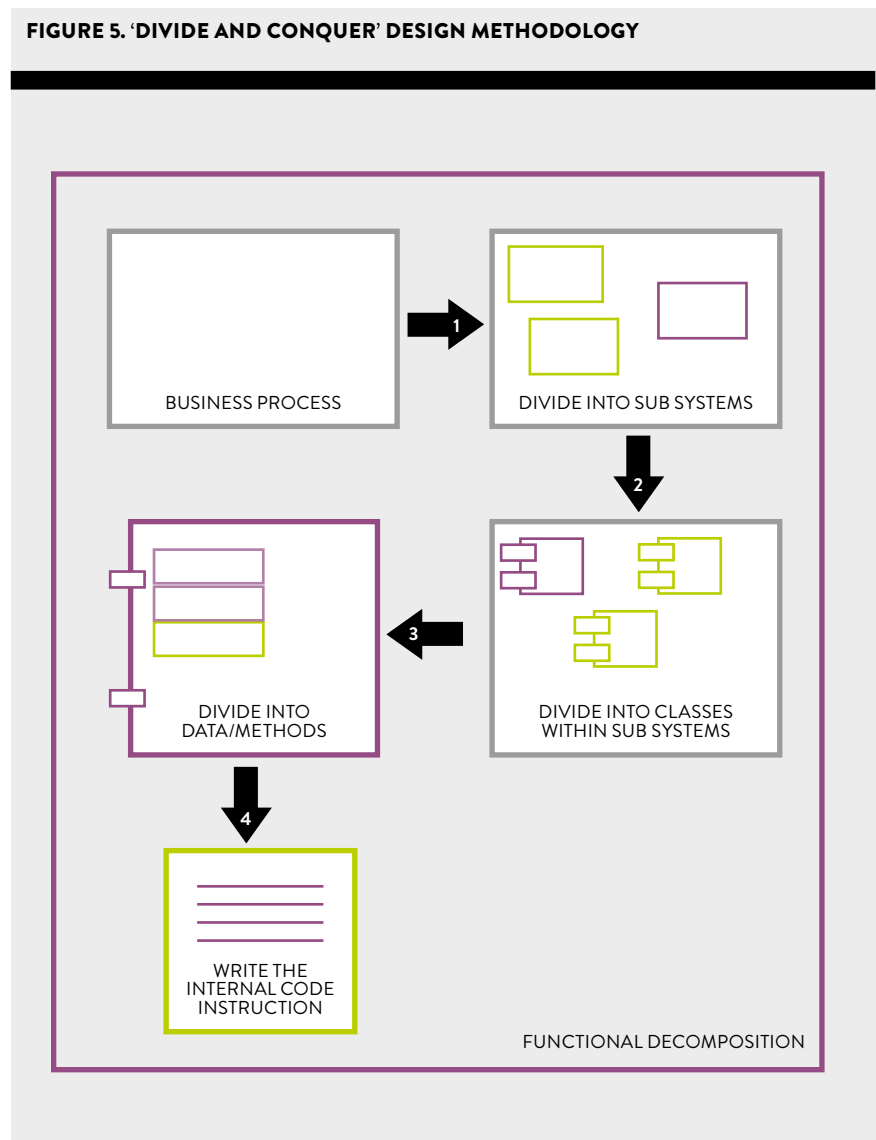
The practice of developers assuming that software can be translated directly from requirements into code does not work. The design process is required as it is difficult to determine precisely from requirements how the solution should work. Design is an iterative, multi-layered approach that describes how the system will be built.

The FDA recognise that **design** is required to 'satisfy the process' and to prevent 'errors in data'⁴⁰. Design is a critical element of the defensive approach as it seeks to reduce complexity⁴¹. Reducing complexity is an important feature of safety and mission critical software⁴². Design is used to minimise the interconnections within software code that propagate negative "ripples" during code fixes. The defensive practitioner will apply a range of techniques in order to reduce the risk of error.

The simplest design methodology is 'divide and conquer'⁴². The system is iteratively decomposed into sub systems and components until distinct functions are identified to contain the business logic. See Figure 5.

The data control flow and defined data is then aligned to components. At each stage, the critical thinking is applied: what will happen if data is changed at this juncture or what roles shouldn't access the system at this point? This approach helps to reduce complexity, support code reuse (to benefit from previously performed quality actions) and facilitate simple and safe maintenance⁴².

FIGURE 5. 'DIVIDE AND CONQUER' DESIGN METHODOLOGY



Key activities that a defensive practitioner should perform include:

- Creating an architectural overview of the proposed system to illustrate how components will relate to each other, what they can and cannot do. A good overview will include alternatives with reasons for selecting the finalised approach⁴²
- Leveraging suitable aspects from robust design methodologies, such as Unified Modelling Language⁴³, to promote process familiarisation, help define the different states that system components can exist in, the data that can be used and the communication between components
- Identifying non-permitted Use Cases and how the system behaves should an unexpected scenario occur
 - Identifying associated error handling routines to provide the correct level of information regarding non-permitted Use Cases
- Performing data flow analysis to focus on data ingestion, data processing and data storage within the system; who and what (function) can access the data; data transfer and how data relates to other data. Focus centres on defining the data and how it is to be managed within the code: such as data type (integer, floating point value, string or array of integers); data file descriptions; database table relationships and so forth. It is useful to define flow to verify data from the terminal, other computerised systems or from data stores
- Conducting system boundary analysis to examine the interaction between software components and other systems with the environment to try and identify vulnerabilities between entities and so help elicit problem conditions. For example:
 - Designing a ‘State Transition Machine’ to ensure that a data item cannot be created or amended until a certain sequence of events have occurred successfully
 - Identifying verification routines to check the processing logic
 - Defining how to communicate data between systems, such as the specific content of data message, message sequencing or failed message delivery
 - Designing the network communications bandwidth to be able to process essential data transfer during high peak usage
- Threat modelling³³ is the specific risk management practice of critically thinking about threats to data integrity and security (Vulnerability Risk Management). Items that may arise from a threat model include:

- Security protocols to ensure that external data is coming from the expected source. For example, using trusted handshakes (digital signatures, certificates) between two systems
- Connecting the computerised system to the managed company IT network to benefit from the shared IT supporting processes. For example, disk replication, access controls.

Design is a critical process and the omission or cavalier performance of this activity, or its review, will result in more defects manifesting in production use.

Front Loading: Additional time spent on early phase activities of robust requirement elicitation and design will result in a multiplier effect on time, effort and budget savings, coupled with increased quality levels later in the validation and production phases.

IMPLEMENTATION STAGE

The implementation stage translates the rules and constraints defined in the design into executable machine code. This is performed by writing commands in a software programming language (code) that is in turn compiled into machine code. The defensive mentality seeks to enforce design and reduce the scope of potential vulnerability issues.

There were over 200 programming languages in active use in 2008³⁴, each with their own specific range of functions. It is important to understand the specific limitations of the programming language in use. Defensive programming will devise software engineering constructs to make up for any language deficits⁴².

Trust but verify: The defensive developer will trust other’s code but will seek to verify that it performs as intended. The adoption of a protective mindset safeguards that others’ code doesn’t harm the data or the underlying business process flow.

The defensive approach **assumes nothing and checks everything**, examples of this critical thinking mindset in action includes:

- Pre-function checks, for example, data value range is in the correct sequence, use of white, grey or black data checklists and so forth
- Initialising variables to prevent reading of previously stored, in memory data
- Setting all return values to negative before processing
- Implementing self-diagnosing code, such as trace statements that write to logs to show what code was executed and when
- Hiding information, such as algorithms, within modules to minimise the impact of ripple effects to other code when logic needs to be changed⁴⁴

- Adding robust error handling to log an error occurrence that facilitates root cause analysis, such as, pass the error statement and value back to a calling component or to gracefully shut down the system⁴²
- Post function checks, for example, return memory to the processor when it is no longer needed, or freeing up connections to a database at the end of processing.

Defensive coding standards include items such as the programming language aspects related to security requirements, software language weaknesses, standard encryption algorithms, interface definitions, how to perform debugging, performance considerations and how to perform robust and effective unit testing⁴².

Test: Defensive unit testing seeks to examine that the design constraints and defensive coding standards have been followed, such as reconciliation routines or record count iterations. Unit tests should also challenge vulnerability constraints such as exception handling, writing to trace files, audit trails, malformed data handling and incorrect loop counters.

Review: The defensive developer embraces code reviews as they are very effective at identifying vulnerabilities and errors, such as missing validation checks, inaccurate logic and concurrency issues. Code reviews are excellent at training junior coders in the practice of defensive programming²⁶.

The defensive practitioner will make use of auxiliary tools to assess confidence in the capability of the software code:

- Static analysis tools to check for code violation, duplicated code, concurrency problems, exposed data definitions and buffer overflows⁴⁵
- Use of code coverage tools to discover which code has not been exercised during unit testing and which test cases are redundant
- Automated compiler checks to check for memory constraints and leaks.

There are a wide range of critical thinking techniques and tools available for the defensive approach.

The simplest critical thinking approach is also the most important: **All code function input data should be considered ‘untrusted’ so all data must be verified**⁴⁶.

‘Code reviews are excellent at training junior coders in the practice of defensive programming.’

INDEPENDENT TEST

An objective of the critical thinking is to reduce the scope of latent vulnerabilities in production that lead to data integrity and security issues.

20 years ago, NIST estimated that a ‘better’ testing infrastructure would save more than \$22 billion per annum⁴⁷. A ‘better’ testing framework involves testing throughout the SLC, from requirements inspections, through to unit testing and on to specialised non-functional testing. For the critical thinker, ‘test early and test often’ starts at the requirements discussion. Regular and repetitive testing seeks to identify defects as early as possible to reduce the risk of fixing ever-increasing interconnected logic at later phases of development. See Figure 6.

Defensive testers need to understand what the software will do and how the software should behave. They will be involved from the outset of the project and engage in the requirements elicitation and review process. Similarly, they will be heavily involved in the design review process.

A defensive test strategy will challenge that the system does what it is supposed to do via static testing (reviewing requirements) or dynamic tests (executing the software):

- Test for each aspect of the design to ensure that it has been correctly implemented
- Test for the data flow within the system
- Check the capability of the logic
- Ensure that the correct data is stored in the expected location in the data store at the correct time of processing. The data store should be checked to categorically demonstrate this.

The critical thinker will then focus on challenging the system so that it does not do what it is not supposed to do (not work as intended⁴⁸).

It is when events occur that were not expected to happen that vulnerabilities will surface. Testing needs to reflect this. The defensive approach is to make the system behave in unpredictable ways and watch how it behaves. For example, challenge the system to ascertain if it can:

- Accept invalid data
- Perform the incorrect sequence of actions
- Deal with missing data values
- Maximise the connection points to the data repository
- Change the data input file type and file
- Apply logically incorrect data
- Amend the database integrity management constraints to force errors into the logic.

(Audit trails would be reviewed during the execution of such test conditions to verify audit trail integrity).

The defensive approach places a strong focus on non-functional testing. Typically, tools are built or bought to force unintended changes in the system. Examples include:

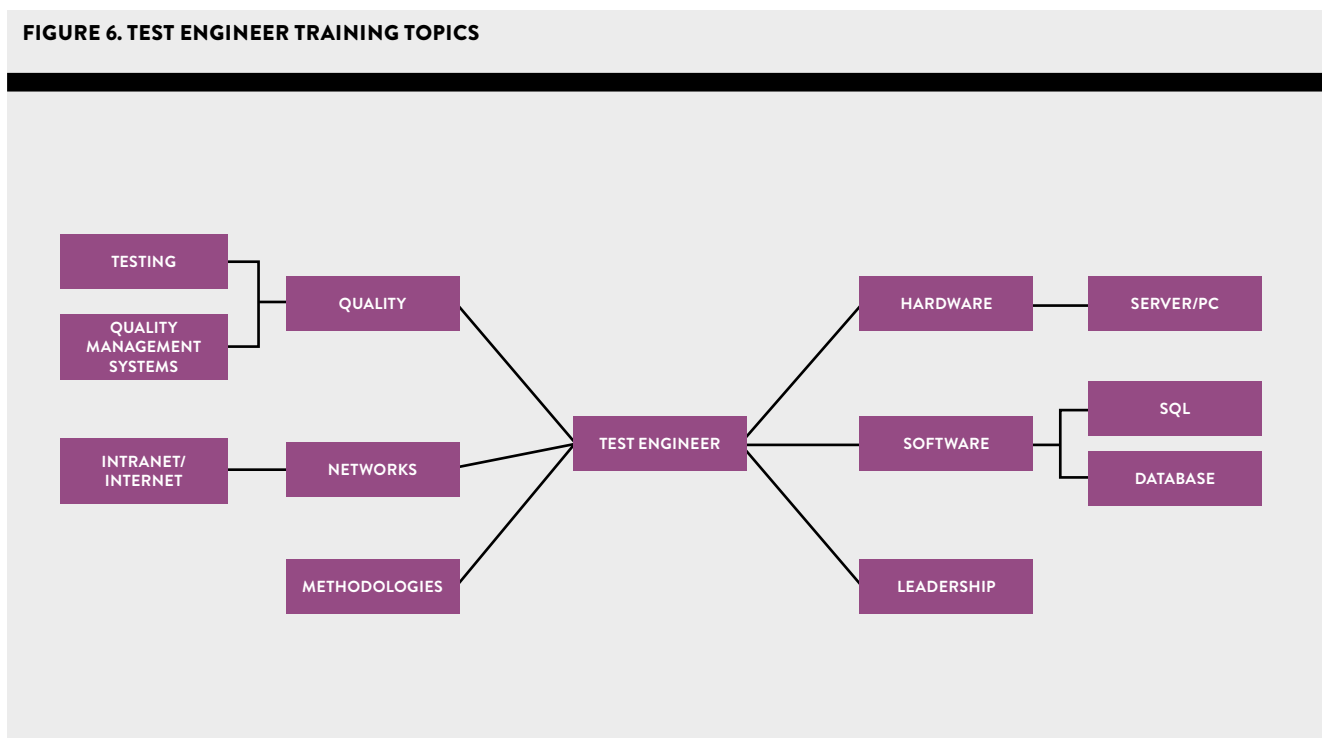
- Maximising processing capacity, system memory and hard disk capacity to try and see how the system copes with hardware stress. Does the system deal with these scenarios correctly, is the data corrupted?
- Performance testing will check how the system constraints can deal with ever-increasing processing demands

- Role-based testing will check how the processing and data will cater for changes in user permissions
- Role-based reviews are performed at the system infrastructure level and may influence the hardening of the computerised system architecture, thereby switching off unnecessary system accounts, processes and communication ports
- Penetration testing challenges running software remotely (as users) to find security vulnerabilities and exploit them. Note that penetration testing is not a panacea: it has been shown to be beneficial for network security but less so for software applications³³
- Security testing checklist³³ to provide simple test cases to be used by testers to challenge exposure to common vulnerabilities, such as buffer overflows, SQL injections, XML and SOAP issues³³.

The defensive practitioner will want to know how effective the SLC quality approach is and whether there are any trends in errors resulting from associated SCL processes that need to be acted upon. Metric collection is necessary to facilitate process improvement.

By capturing metrics at each phase (requirement inspections metrics, design review metrics, various testing metrics) the defensive practitioner will be able to assess the adequacy of the approach and establish if quality is being built in. An indicator of a mature and effective supplier will be their ability to accurately predict the level of quality in operational use³⁴.

FIGURE 6. TEST ENGINEER TRAINING TOPICS





SUMMARY

“The problem of insecure software is perhaps the most important technical challenge of our time”³³.

The economic focus on IT infrastructure protection and monitoring reflect the author’s audit discussions with IT departments on data integrity and security. These discussions reveal that the same focus may have been marginalised when it came to securing software.

This article discusses the defensive approach to building secure software for computerised systems and briefly mentions some of the common critical thinking techniques and processes readily available for use within the SLC.

No single technique will offer full immunisation to data integrity ills. It takes an amalgamation of ‘appropriate’ SLC techniques to build data quality in and limit the window of opportunity for data and system compromise. Evidence of these practices during supplier audits should indicate a better quality and security-focused supplier.

A defensive approach will incur additional (although relatively minor) upfront cost and effort at the early phases of a project, but by reducing the scope of defects being designed and built into a computerised system, it will yield the following benefits:

- Enhanced compliance to regulatory expectation with greater SLC process control⁴¹
- Reduced defects in software, e.g. 45% reduction in XP/Vista vulnerabilities after one year⁴⁹
- Reduced software vulnerabilities with a reduction in the breadth and depth of defects⁵⁰
- Hardened (secure) software to mitigate against IT system breaches
- Reduced costs from enhanced quality, for example, HP’s average time to fix defects fell from two weeks to two hours⁵⁰
- Faster time to production as there is less defects to correct, release, retest and document
- Overall lower project costs as a result³⁴.

REFERENCES

1. 21CFR Part 11 (11.10), FDA, Jan 2022
2. MHRA Data Integrity Guidance, MHRA, March 2018
3. 21CFR Part 211 Sub-part D, J, FDA, Jan 2022
4. 21CFR Part 820 (72.2) (180), FDA, Jan 2022
5. FDA Guidance for Industry, Computerized Systems used in Clinical Trials, FDA, April 1999
6. Notice to sponsors on validation and qualification of computerised systems used in clinical trials, EMA, 07 April 2020
7. Essential Management Information Systems, p231, Laudon & Laudon, 8th Edition, Pearson Prentice Hall, 2009
8. Gartner Forecasts Worldwide Security and Risk Management Spending to Exceed \$150 Billion in 2021, Gartner Newsroom release, January 2022
9. National Vulnerability Database, NIST, 15-June-2020
10. Vulnerabilities definition, National Vulnerability Database, NIST, 01 Feb 2022
11. Yes, terrorists could have hacked Dick Cheney’s heart, Washington Post, Oct 2013
12. Healthcare Ransomware Attacks Accounted for 50% of All Security Incidents, spamtitan.com, April 2017
13. Merck & Co. Becomes Victim of Massive Ransomware Cyber Attack, BiosSpace.com, June 2017
14. Major breach found in biometrics system used by banks, UK police and defence firms, Guardian, Aug 2019
15. Lifelabs Data Breach, the Largest Ever in Canada, May Cost the Company Over \$1 Billion in Class-Action Lawsuit, Scott Ikeda, CPO Magazine, January 2020
16. Equinix data center giant hit by Netwalker Ransomware, \$4.5M ransom, Lawrence Abrams, Bleeping Computer, September 2020
17. Dr Reddy’s isolates data centre services after cyber attack, Prabha Raghavan, The Indian Express, February 2022
18. Guideline on data integrity. WHO Drug Information, 33 (4), 773 - 793. World Health Organization, 2019.
19. FDA Warning Letter FDA, 17-August-2021, FDA, Aug-2021
20. FDA Warning Letter 08 July 2020, FDA, Jul-2020
21. PwC 2022 Global Digital Trust Insights, Sean Joyce, PWC, 2021
22. Gartner Says More than 75 Percent of Mobile Applications will Fail Basic Security Tests Through 2015, Gartner Newsroom, 14 September 2014
23. 92% of External Web Apps Have Exploitable Security Flaws or Weaknesses: Report, Security Week, 30 October 2018
24. Cyber-Attacks to Devices Threaten Data and Patients, IEEE PULSE, Jan 2018
25. Cybercrime and Other Threats Faced by the Healthcare Industry, Trend Micro, 2017
26. Over 50% of internet-connected medical devices vulnerable to cyberattacks, John Fischer, Health Care Business, January 2022
27. Boy, 11, hacks cyber-security audience to give lesson on ‘weaponisation’ of toys, Guardian, May 2017
28. How a fish tank helped hack a casino, Washington Post, Jul 2017
29. How to get rich in Silicon Valley, Guardian, 17-Apr-2018.
30. The log4j security flaw could impact the entire internet. Here’s what you should know, Jennifer Korn, CNN Business, December 2021
31. Cost of a Data Breach Report 2021, IBM, July 2021
32. Tools of Critical Thinking, Levey, Waveland Press, 2003

33. OWASP Testing project 4.0, chapter 2, page 12, OWASP 2013
34. Applied Software Measurement, Capers Jones, 3rd edition, McGraw Hill, 2008
35. What is a Software Vulnerability? JFrog Support, Aug-2022
36. ‘GXP’ Data Integrity Definitions and Expectations, MHRA, 2018
37. ISPE GAMP Records and Data Integrity GPG Data Integrity – Key Concepts, Appendix 6 Requirements Planning, ISPE, 2018
38. I keep Six Honest Serving Men, Rudyard Kipling, circa 1900
39. Guide for Conducting Risk Assessment, NIST, Sep-2012.
40. Guidance for Industry, Computerized Systems Used in Clinical Investigations FDA, May 2007
41. General Principles of Software Validation, FDA, 2002
42. Code Complete 2, A Practical Handbook for Software Construction, 2004, Steve McConnell, Microsoft Press.
43. OMG® Unified Modelling Language® (OMG UML®) Version 2.5.1, Object Management Group, 2017
44. Improving Software Productivity, in Computer, vol. 20, 1987.
45. Static Code Analysis, OWASP, (Website in transition with date not published)
46. Writing Secure Code, Howard, LeBlanc, Microsoft Press, 2002.
47. The economic impacts of inadequate infrastructure for software testing, NIST, MAY 2002
48. “Negative Testing” definition, Boris Beizer, The Testing Standards, version 6.3 BCS SIGIST
49. Windows Vista One Year Vulnerability Report, Microsoft Security Blog 23, Jan 2008, referenced in: Introduction to the Microsoft Security Development Lifecycle SDL, Summary PowerPoint.
50. Does Application Security Pay? Measuring the Business Impact of Software Security Assurance Solutions, Mainstay Partners, 2020

PROFILE

Barry has over 25 years’ experience in IT solutions, with vast technical knowledge of every role and every activity within the Computerised System Life Cycle. He transitioned into the regulated industry in 2003 to successfully build bespoke SLC/Validation lifecycles that yielded right first time validation and started Empowerment Quality Engineering in 2007 to apply technical and quality knowhow to the wider pharmaceutical industry, from IT SOPs and supplier assessments to validation projects. Barry is a member of the RQA IT Committee and is a tutor on the Computer System Validation course and Auditing Computerised Systems course. He is a member of IEEE and ISPE, authoring Appendix 6 (Requirements Elicitation) of the Data Integrity Good Practice Guide.