**Software Vulnerabilities, Prevention and Reduction**

**Methods: Assessment**

Matthew Sides

West Herts College, Watford

Hertfordshire, England

Last Update: 25/02/2018

19/02/2018

**Abstract.**

Through analysis and deduction, this report contains information based around the topic of software development and vulnerabilities as a whole, further delving into the idea of vulnerabilities in software development through case study examples. Subsequently further down the document the mitigation of said vulnerabilities is covered, explaining a few methods of preventing or fixing said breaches that may occur, coming to a conclusion as to whether the case studies overlooked could have been mitigated and prevented before the situation spiralled out of hand.

**Keywords:** Software Development, Vulnerabilities -

Mitigation, Testing, Detection and Prevention

**Table of Contents**

[1] Introduction

[2] Software Development Topic

[3] Vulnerabilities Topic

[4] Case studies

[5] Mitigation of Vulnerabilities

[6] Conclusions

[7] Bibliography

**Introduction [1]**

This documentation pertains to the reduction of software vulnerabilities during development, further delving into the subject or topics of what software development (this includes Software-Development-Life-Cycles | SDLC) and vulnerabilities are, defining and suggesting possible solutions to said vulnerabilities during development, using case studies as illustrations and to portray the risks that herald with not mitigating vulnerabilities properly.

Thereafter concluding the various ways in which to resolve (fix) and reduce vulnerabilities encountered during the development or finished production of a system, covering a self-opinionated view and lightly addressing the issues usually encountered, re-defining software development and vulnerabilities to better bring the final point across (use as a reminder).

**Software Development Topic [2]**

Software development is a meticulous and complex process to develop software, comprised of numerous stages. Each stage requires tentative documentation in addition to the development and planning process, in contrast to what is commonly known or how software development is usually portrayed (in media) Jimenez et al. No date).

A software development project must at least entail the following stages (though not constrained to the below list), a view supported by Abrahamsson et al. (2002):

1. Requirement gathering
2. Writing functional specifications
3. Creating architecture and design documents
4. Implementation and coding
5. Testing and quality assurance
6. Software release
7. Documentation
8. Support and new features

**Requirement Gathering [2.1]**

The act of requirement gathering essentially pertains to the act of asking or requesting that clients give brief based information concerning the reason for employment and said “goals and achievements” that the client expects from the company.

Moreover, clients or prospective clients during this phase or stage usually analyse market requirements and features that are in demand. Further delving into the supply and demand aspect of the market seeing whether there is a real need for the software that is being developed in the current market.

A comprehensive understanding of the customers’ needs and writing down features of the proposed software product are the keys to success in this phase.

Ur Rehman (2001) agrees that this phase is fundamentally the foundation or base for a project and failure to uphold or properly document could lead to a monumental catastrophe, with the project either failing to come together or veering off the client’s direction and vision for the project, not withholding or meeting the standards that the client may expect.

Consequently, the opposite applies with regards to the current market and economic position, if a project is pushed through and is deemed an excellent software product, as long as it is not required in the market, it is most likely to be unsuccessful.

**Writing Functional Specifications [2.2]**

Following said client gathering, a review should ensue to determine whether said pre-defined or set requirements given by the client are plausible, subsequently then determining as to how to achieve said specifications (Functional specifications are based upon the product requirements documentation put forward), drawing upon a functional specification document, showing the behaviour or functionality of a software product on an abstract level.

The prominence of the usage of writing functional specifications relate to the creation of a design document, manuals for end-users, functional and architecture documents, being a means to communicate among groups disconnected by their software development landscape. Although during the product development phase amendments to functional specifications may be necessary, keeping in view new marketing requirements.

**Architecture and Design Documents [2.3]**

The architecture document in effect is formed once all the requirements have been collected and arranged. The document defines different components of the product and how they interact with each other.

Further defining the technologies used to build the product. Whilst the documents are filled out and moulded around the project, the (project) team also needs to consider the timelines of the project. This refers to the target date when the product is required to be on the market.

Thereafter defining architecture, software components and their interfaces, the next phase of development is the creation of a design document. Through the architecture level, a component is defined and stated to provide certain functionality. Whilst during the generation of the design document (stage), you have to define what is in the component.

Although the design document and architecture document could be viewed or merged into one fully fledged documentation to easier navigate and locate information based on the project and formulate all the information on a component in one space rather than separating said information, hence while although the architecture and design are seen as two separate levels, the design document is essentially just elaborating on the architecture level, adding a miniscule amount of information still pertaining to the component only rather than for instance describing more explaining said component.

**Implementation and coding [2.4]**

The Implementation and coding phase is initiated when software developers take the design documents and development tools (editors, compilers, debuggers etc.) and start writing software (Halvorsen 2017).

This is an erroneous process and usually the longest phase in the product life cycle. Each developer has to write their own code to a standard, whilst collaborating with other developers to ensure that different components can interoperate with each other.

Subsequently this problematic issue may-be generally bypassed through using or outlining coding standards beforehand, ensuring that the written procedures are followed or abided by, thus enabling the code to be similar structurally and aesthetically. Furthermore, elaborating the reason for coding standards, during the project or product life cycle multiple people may modify files, therefore if people are not following the pre-set rules, this may easily break the whole compilation and building process. For instance, duplicate definitions of the same variables may cause problems, while similarly, if included files are not written properly, you can easily cause the creation of loops.

**Testing and quality assurance [2.5]**

Subsequently once the implementation and coding stage is completed, the next phase in software development is to test said code to see whether it is functional and works to an exceptional standard meeting and assuring the quality. This may be initiated through managing the product life cycle and verifying that the software meets the defined quality standards or customer agreements (quality assurance), thereafter with regards to the testing aspect may be instigated during the production phase, concerned with finding ways of breaking software, coding in segments or pushing out versions of said application software are integral to ensure a more accurate test.

In addition, throughout testing the ideology behind it is to get a group of testers to observe what the software does and to report on the level of quality as well as any serious issues they encounter (under the assumption that the system is flawed and has copious amounts of bugs) (Qatestlab Blog 2005-2010).

**Software release [2.6]**

Before the subsequent marketing and selling of a software product it needs to be officially released. This essentially pertaining to the creation of a state of the software in a repository (a copy), ensuring that it has been tested for functionality, etc. A version number is assigned to the released software. After releasing the software, development may continue but will not impact or make any subsequent changes to the released software. The development is usually carried on in a new branch and it may contain new features of the product. The released software is updated only if a bug fixed version is released.

Usually incremental version numbers are assigned following a scheme of when the next release of the software is sent to market, whilst the change in version number depends on whether the new software contains a major change to the previous version or it contains bug fixes and enhancement to existing functionality. Releases are also important because they are typically compiled versions of a particular version of the code, and thus provide a stable set of binaries for testing (In simpler terms the paragraphs above pertain to the updates and patches in software development after and before release).

**Documentation [2.7]**

**Reliable documentation is crucial throughout software development (life cycle).** The presence of documentation helps keep track of all aspects of an application and improves on the quality of a software product. Furthermore, mainly focusing on **development, maintenance and knowledge transfer**to other developers (Trica 2017).

Through the use of said documentation, information should be more easily accessible (information all gathered in one point or origin). Moreover, the use of documentation provides a limited number of user entry points, help new users learn quickly, simplify the product and help cut support costs.

What is more the documentation is usually focused on the following components that make up an application: s**erver environments, business rules, databases/files, troubleshooting, application installation and code deployment.**

**Support and New Features [2.8]**

This usually heralds after the development of software, though is still integral to the cycle as users of said developed software may need support after release. In addition, new features may be a factor or prerequisite with updates coming out to compete with competitors’ products that may have other features. Moreover, better support usually leads to better branding and devotion among user’s, creating or leading to the generation of a referral business.

**Software Development Life Cycles [2.9]**

SDLC or the abbreviated term System Development Life Cycle is a term generally applied to system engineering, information systems and software engineering to describe the process for planning, creating, testing, and deploying an information system (Wilkipedia 2018).

This interconnects with software development as the numerous life cycle’s methodology’s define the procedure’s and show regulated development, portraying the process of the plan, creation and deployment of an information system.

Further once applied to the development of software acts as a formatted structure, that once followed may bring about advantages or wrought disadvantages depending on the life cycle used, all singular life cycles are useful if applied in the correct situations and therefore it is necessary to analysis the project size, structure, requirements, etc. before a life cycle is chosen to be applied.

Examples of software development life cycles consist of; water fall method a sequential non-iterative design model, used in software development processes.

Moreover, as suggested in the conceptual name and illustrated in the overall outlook of the theoretical waterfall model it can be likened to that of a literal waterfall, following a similar system; flowing steadily downwards (hence the name waterfall) through phases of conception, initiation, analysis, design, construction, testing, production/implementation and maintenance.

In addition to the waterfall methodology or life cycle there is also the Prototyping Model is a systems development method (SDM) in which a prototype (an early approximation of a final system) displays the functionality of the product under development, but may not actually hold the exact logic of the original software (early copy of the system, used to understand the requirements). That is to say that it is built, tested, and then reworked as necessary until an acceptable prototype is finally achieved from which the complete system or product can now be developed (the prototype is essentially the frame work) (Sides 2017).

Concurrently there is also the Evolutionary Model or approach, based on the idea of rapidly developing an initial software implementation from very abstract specifications and modifying this according to an appraisal (phases are concurrent and feedback is given and used between each phase).

Further examples may be extracted from Alan (2017), as there are numerous other cycles, though not covered in this report, since the above section is merely giving examples to better illustrate the diversity of software development lifecycles, explaining SDCL as a whole and there or its purpose in the development of software (software development).[[1]](#footnote-1)

**Vulnerabilities Topic [3]**

The topic of Software vulnerabilities pertains to the (weaknesses/flaws) faults that may be exploited within a given system, thereafter compromising the system itself. A systems flaws may be subjugated or subjective to various different types of vulnerabilities and thereafter it is crucial to know the diverse array of vulnerabilities, their prevention and detection in order to try to avoid their presence in the final software version of the system and then reduce the possibility of attacks and costly damages.

The knowledge of any given vulnerability in a system may give an attacker who is exploiting said vulnerability the ability to reduce a systems information assurance, also allowing access to sensitive files and data harmful to the cliental of an organisation (user’s) or the organisation itself (exploiting a loophole).

Vulnerabilities are essentially the intersection of three elements: a system susceptibility or flaw, attacker access to the flaw, and attacker capability to exploit the flaw (Jimenez et al. no date).

**Examples of Vulnerabilities [3.1]**

Following the above documentation, under the heading “Vulnerabilities Topic”, listed with the numerical digit 3, below details a few frequent vulnerabilities (Jimenez et al. no date):

* Buffer overflow: this usually occurs during fixed length buffers when some data is going to be written beyond the boundaries of the current defined capacity (hence the use of the word “overflow”).

This could lead to the system malfunctioning since the new data could in principle corrupt the data of other buffers or processes.

In continuation Buffer overflow could also lead to loss of control and gain (for an attacker) over a system, through the injection of malicious code, subsequently altering the execution sequence of the program in order to execute the injected code, thus taking control of the system.

* XSS or cross site scripting: commonly associated or connected to web applications, cross site scripting consists of the injection of code in pages accessed by other users. Through exploitation an attacker may bypass access controls, perform phishing, identity theft or expose connections.
* SQL injection: consists of injecting code with the intension of exploiting the content of a database. This usually happens because the inputs are not handled correctly, an attacker may retrieve or get sensitive information from the database (under attack).

**Case Studies [4]**

**Case Study:” goto fail” 2014 Apple [4.1]**

In 2014, Apple fixed a critical security vulnerability that was sought to be caused through the use of "cut and paste" during editing. The programmer embedded a duplicate line of code that caused the software to bypass a block of code that verifies the authenticity of access credentials (connections to secure sites could be spied on). The essence of the problem, though straight forward, led to invalid certificates being quietly accepted as valid (woody 2015).

The vulnerability is described in the National Vulnerability Database cited by Ducklin (2014) as follows:

* "Impact: An attacker with a privileged network position may capture or modify data in sessions protected by SSL/TLS" –
* "Description: Secure Transport failed to validate the authenticity of the connection. This issue was addressed by restoring missing validation steps." -

Researchers learnt of this security flaw in iPhones and iPads, with the confirmation of Apple stating that it was also present in notebook and desktop machines using the Mac OS X operating system.

This vulnerability essentially led to attackers being able to use invalid credentials to gain access to any information on the targeted device such as email, financial data, and access credentials to other systems and devices.

Further elaborating on the root cause of the problem the second (duplicate) “goto fail, the indentation found within the code (segment displayed below) was misleading, since the code didn’t retain or use curly braces after the “if” statement, the second “goto fail” was therefore always executed.

In context, this meant that vital signature checking code was skipped, so all signatures regardless of authenticity were registered as acceptable. The extraneous “goto” caused the function to return 0 (“no error”) when the rest of the checking was skipped, as a result causing invalid certificates to be accepted and thus lead to what is now termed or coined as the "goto fail" incident (Wheeler 2017).

**[4.2] Case Study: 2017 Equifax Breach**

In 2017 Equifax experienced a breach in their systems resulting in the loss of sensitive data and personal records on 145 million Americans and hundreds of thousands in the UK and Canada. The extent and impact of such a breach, along with the information procured by attackers could have led to identity theft (impersonation), as seen in similar cases earlier experienced in 2017, known as "[IRS FAFSA tool breach](https://www.nytimes.com/2017/04/06/us/politics/internal-revenue-service-breach-taxpayer-data.html)" (Qatestlab Blog 2005-2010).

Further delving into how the breach resulted and vulnerabilities was exploited, it occurred due to the negligence of Equifax, failing to find and promptly patch publicly known vulnerability in Apache Struts (a free, open-source, MVC framework for creating elegant, modern Java web applications.), which was used in its Consumer Dispute Portal.

In simple terms apache struts was essentially used to develop their own software or portal that held important user information, however during development and due to a fault in the software used for development (apache struts) a vulnerability materialized and went unnoticed.

Moreover, the vulnerability was exploited over 2 months after a fix was made available. In addition, the company did not discover that its systems had been compromised for another 1.5 months. This series of failures demonstrates the significant weaknesses in Equifax’s security program during this time frame (Sethi 2017) (BBC News 2017) (Thomson 2018).

**Mitigating Software Vulnerabilities [5]**

Models are a first approach to deal with vulnerabilities and their understanding (putting in stronger controls and fleshing out the software better). However, it is necessary to count on methods or procedures to prevent any risks related to vulnerabilities.

Further ways of mitigation is software inspection, this usually consists of reading or visually inspecting the program code or documents in order to find any defects and correct them early in the development process, though this generally relies on the ability of the person viewing the code (Simmons 2015).

Security testing is also another integral method of finding vulnerabilities in a system, if the vulnerabilities make it all the way to a test environment, security testing tools and manual techniques may discover these types of issues (Leemon 2015).

In regards to mitigating or preventing software vulnerabilities being exploited, that have occurred during the development of software but went unnoticed, consistent monitoring of applications for unusually activity may halt attacks, though not ideal in this phase, it is critical to have controls in the right place (National Cyber Security Centre 2016).

**Conclusion [6]**

In conclusion forming an opinion on the basis of the information presented above, concerning the case studies, in the Apple situation and Equifax state of affairs, during both occurrences the vulnerabilities presented and exploited by the alleged attackers could have been prevented and mitigated if the necessary cautions and procedures were put in place and followed, with the apple incident stemming from human negligence that could have been prevented if a proper software inspection took place , similarities can be seen with regards to the Equifax situation, where if the code was manually reviewed, steps could have been put in motion to mitigate said vulnerabilities that appeared.

Through further analysis it may be seen that the response taken by Apple was optimal, though with regards to Equifax's response not all information regarding the attack was exposed and neither were there consistent monitorisation by their operation team as the exploitation of said vulnerability had occurred 1.5 months before the organisation had found out about the attack , showing that the operation team was not searching for unusual activity which could have led to the situation devolving before reaching the stage it did. Whilst it also took another 2 months to fix and roll out a patch.

In my opinion, which is supported by research (Anton et al 2003) whilst the situation for both high profile organisations were eventually sorted with the vulnerabilities found in the condescending software being mitigated. The foremost factor being whether these's vulnerabilities could have been mitigated earlier (during development), rather than only after release. Which based on factual evidence and methodologies used in the prevention of vulnerabilities, points to the decisive answer being certainly.

**Bibliography [7]**

* Abrahamsson, P., Salo, P., Ronkainen, O. and Warsta, J. (2002). *Agile Software Development Methods: Review an Analysis*. VTT Publications. Available at: [https://arxiv.org/ftp/arxiv/papers/1709/1709.08439.pdf](https://arxiv.org/ftp/arxiv/papers/1709/1709.08439.pdf%20) [Accessed on ????]
* Antón, Phillip., Anderson, Robert., Mesic, Richard., Scheiern, Michael. (2003) *Finding and Fixing Vulnerabilities in Information Systems. The Vulnerability, Assessment & Mitigation Methodology*. Santa Monica: RAND.
* Allen, Paul.G School of Computer Engineering and Science. (2017). *Software development lifecycle.* Available at: <https://courses.cs.washington.edu/courses/cse403/16sp/lectures/lecture-03-software-lifecycle.pdf> [Accessed on:????]
* BBC News. (2017). *Equifax data hack affected 694,000 UK customers*. Available at: <http://www.bbc.co.uk/news/business-41575188> [Accessed on:??????]
* Duckins, Paul. (2014). *Anatomy of a “goto fail” – Apple’s SSL bug explained, plus an unofficial patch for OS X!* Available at: [https://nakedsecurity.sophos.com/2014/02/24/anatomy-of-a-goto-fail-apples-ssl-bug-explained-plus-an-unofficial-patch/](https://nakedsecurity.sophos.com/2014/02/24/anatomy-of-a-goto-fail-apples-ssl-bug-explained-plus-an-unofficial-patch/%20) [Accessed on:????]
* Jimenez,W., Mammar, A., and Cavalli, A., (No date). *Software Vulnerabilities, Prevention and Detection Methods: A Review1*. Available at: [http://www-lor.int-evry.fr/~anna/files/sec-mda09.pdf](http://www-lor.int-evry.fr/~anna/files/sec-mda09.pdf%20) [Accessed on:????]
* Halvorsen, H-P. (2017)*. Software Development: A Practical Approach!* Available at: [http://home.hit.no/~hansha/documents/software/software\_development/resources/Software Development.pdf](http://home.hit.no/~hansha/documents/software/software_development/resources/Software%20Development.pdf) [Accessed on:????]
* Leemon, Alex. (2015). *5 IT BEST PRACTICES THAT ALSO MITIGATE CYBER SECURITY VULNERABILITIES IN OT.* Available at: <https://www.cyberark.com/blog/5-it-best-practices-that-also-mitigate-cyber-security-vulnerabilities-in-ot> [ Accessed on: ????]
* National Cyber Security Centre. (2016). *Common cyber attacks: reducing the impact.* Available at: <https://www.ncsc.gov.uk/content/files/protected_files/guidance_files/common_cyber_attacks_ncsc.pdf> {Accessed on: ????]
* Qatestlab Blog (2005-2010). *Browser Update*. Available at: <http://blog.qatestlab.com/2011/04/07/what-is-the-difference-between-qa-and-testing/> [Accessed on:????] (blog.qatestlab)Rappeport, A. (2017). *Up to 100,000 Taxpayers Compromised in Fafsa Tool Breach, I.R.S. Says.* The New York Times. Available at: <https://www.nytimes.com/2017/04/06/us/politics/internal-revenue-service-breach-taxpayer-data.html> [Accessed on: ???]
* Thomson, Iain. (2018). *Equifax hack worse than previously thought: Biz kissed goodbye to card expiry dates, tax IDs etc.* Available at: <https://www.theregister.co.uk/2018/02/13/equifax_security_breach_bad/> [Accessed on: ????]
* Sethi, Ami. (2017). *What can your firm learn from the unfolding Equifax hack?* Available at: <https://www.synopsys.com/blogs/software-security/security-breach-prevention-tips/> [Accessed on:????}
* Sethi, Amit. (2017). *Top security breaches of 2017 (+2018 cyber security predictions).* Available at: <https://www.synopsys.com/blogs/software-security/top-data-security-breaches-2017/> [Accessed on: ????]
* Sides, Matthew., (2017). Available at: <https://github.com/matthewsides/SDCL-Methodologies> [Accessed on: ????]
* Simmons, Rod. (2015). *4 Steps to Mitigate 95% of Known Vulnerabilities*. Available at: <http://www.itprotoday.com/security/4-steps-mitigate-95-known-vulnerabilities> [Accessed on: ???]
* Trica, Alex. (2017). *The Importance of Documentation in Software Development.*  Available at: <https://filtered.com/blog/post/project-management/the-importance-of-documentation-in-software-development> [Accessed on: ????]
* Wheeler. David. (2017). *The Apple goto fail vulnerability: lessons learned* *.* Available at: <https://www.dwheeler.com/essays/apple-goto-fail.html> [Accessed on: ?????]
* Wilkipedia (2018). *Systems development life cycle*. Available at: <https://en.wikipedia.org/wiki/Systems_development_life_cycle> [Accessed on: ?????]
* Woody, Carol. and Nichols, Bill. (2015). *Heartbleed and Goto Fail: Two Case Studies for Predicting Software Assurance Using Quality and Reliability* *Measures.* Available at:[https://insights.sei.cmu.edu/sei\_blog/2015/04/heartbleed-and-goto-fail-two-case-studies-for-predicting-software-assurance-using-quality-and-reliab.html](https://insights.sei.cmu.edu/sei_blog/2015/04/heartbleed-and-goto-fail-two-case-studies-for-predicting-software-assurance-using-quality-and-reliab.html%20) [Accessed on: ????]
* Ur Rehman, R., Paul, C. (2002). *The Linux Development Platform: Introduction to Software Development Chapter 1.* Available at: <http://catalogue.pearsoned.co.uk/samplechapter/0130091154.pdf>

1. [↑](#footnote-ref-1)