SET10113 – Secure Software Development

Part 1

1.Access Control

In order to fix the access control implementation, the ABAC strategy has been implemented adding the *@PreAuthorize* annotation to each of the classes that should be accessible just for the Administrator of the portal.

- *CourseResource*

```
36   */
37  @RestController
38  @RequestMapping("/api")
39  @PreAuthorize("hasRole('ADMIN')")
40  public class CourseResource {
41
```

The required annotation has been added to the */home/student/Secure-Software-Development-Module-Coursework/NapierUniPortal/src/main/java/uk/ac/napier/soc/ssd/coursework/web/rest/CourseResoure* Class. Adding it on the top of the *CourseResource* Class will not allow the User neither View, Add, Edit or Delete Courses.

- *ProgramResource*

```
54    */
55   @PostMapping("/programs")
56   @Timed
57   @PreAuthorize("hasRole('ADMIN')")
58   public ResponseEntity<Program> createProgram(@Valid @RequestBody Program program) throws URISyntaxException {
59       log.debug("REST request to save Program : {}", program);
60       if (program.getId() != null) {
```

```
79   @PutMapping("/programs")
80   @Timed
81   @PreAuthorize("hasRole('ADMIN')")
82   public ResponseEntity<Program> updateProgram(@Valid @RequestBody Program program) throws URISyntaxException {
83       log.debug("REST request to update Program : {}", program);
84       if (program.getId() == null) {
```

```
125  @DeleteMapping("/programs/{id}")
126  @Timed
127  @PreAuthorize("hasRole('ADMIN')")
128  public ResponseEntity<Void> deleteProgram(@PathVariable Long id) {
129      log.debug("REST request to delete Program : {}", id);
130
```

Along with the previous file, the required annotation has been added to the */home/student/Secure-Software-Development-Module-Coursework/NapierUniPortal/src/main/java/uk/ac/napier/soc/ssd/coursework/web/rest/ProgramResource* Class. In this case because the user should be able to View the programs, the annotation couldn't added to the top of the class and it has to be added three times, one to each of the functions to each task that user is not allow to do.

- *EnrolmentsResource*

```
127         Optional<Enrollment> enrollment = enrollmentRepository.findOneWithEagerRelationships(id);
128
129         if(!SecurityUtils.getCurrentUserLogin().orElse( other: "").equals("admin") && enrollment.isPresent() &&
130             enrollment.get().getUser() != null &&
131             !enrollment.get().getUser().getLogin().equals(SecurityUtils.getCurrentUserLogin().orElse( other: ""))){
132             return new ResponseEntity<Enrollment>(HttpStatus.FORBIDDEN);
133         }
134         return ResponseUtil.wrapOrNotFound(enrollment);
135     }
```

The same annotation has been added to the */home/student/Secure-Software-Development-Module-Coursework/NapierUniPortal/src/main/java/uk/ac/napier/soc/ssd/coursework/web/rest/EnrolmentsResource* class. This check if the *CurrentUserLogin* is the same as the one in the Enrolment. If it is not it does allow the user to see the information.

```
112     public List<Enrollment> getAllEnrollments(@RequestParam(required = false, defaultValue = "false") boolean eagerload) {
113         log.debug("REST request to get all Enrollments");
114
115         if(SecurityUtils.getCurrentUserLogin().orElse( other: "").equals("admin"))
116             return enrollmentRepository.findAllWithEagerRelationships();
117         else
118             return enrollmentRepository.findByUserIsCurrentUser();
119     }
120
```

In addition to the previous change. The getEnrollment class has been modified so users can only see the enrolments they are involved in.

```
public ResponseEntity<Enrollment> getEnrollment(@PathVariable Long id) {
    log.debug("REST request to get Enrollment : {}", id);
    Optional<Enrollment> enrollment = enrollmentRepository.findOneWithEagerRelationships(id);

    if(!SecurityUtils.getCurrentUserLogin().orElse( other: "").equals("admin") && enrollment.isPresent() &&
        enrollment.get().getUser() != null &&
        !enrollment.get().getUser().getLogin().equals(SecurityUtils.getCurrentUserLogin().orElse( other: ""))){
        return new ResponseEntity<Enrollment>(HttpStatus.FORBIDDEN);
    }
    return ResponseUtil.wrapOrNotFound(enrollment);
}
```

Beside that change, the required annotation has been added as in the previous classes that will not allow the user to neither Create, Edit or Delete any enrolment.

```
58      @PostMapping("/enrollments")
59      @Timed
60      @PreAuthorize("hasRole('ADMIN')")
61      public ResponseEntity<Enrollment> createEnrollment(@RequestBody Enrollment enrollment) throws URISyntaxException {
62          log.debug("REST request to save Enrollment : {}", enrollment);
63          if (enrollment.getId() != null) {
```

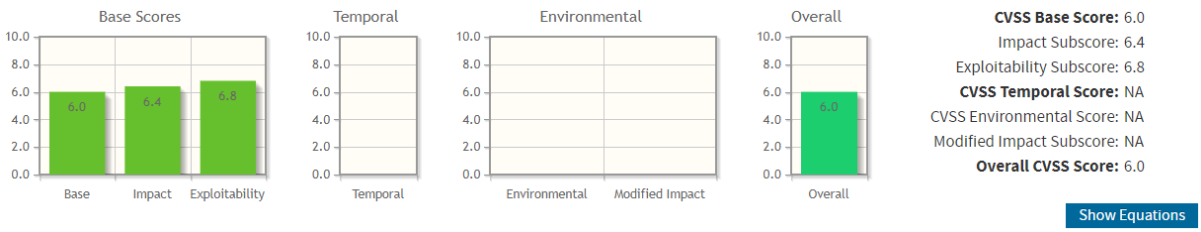
```
151     @GetMapping("/_search/enrollments")
152     @Timed
153     @PreAuthorize("hasRole('ADMIN')")
154     public List<Enrollment> searchEnrollments(@RequestParam String query) {
155         log.debug("REST request to search Enrollments for query {}", query);
156         Session session = HibernateUtil.getSession();
```

```
83      @Timed
84      @PreAuthorize("hasRole('ADMIN')")
85      public ResponseEntity<Enrollment> updateEnrollment(@RequestBody Enrollment enrollment) throws URISyntaxException {
86          log.debug("REST request to update Enrollment : {}", enrollment);
87          if (enrollment.getId() == null) {
```

2. Identifying and fixing XSS vulnerabilities

The two vulnerabilities have been found with manual review. One of them was in *EnrollmentResource* and the other one in the *course-detail.component.html*.

Both are crucial because they allow to insert JavaScript code into the database which can allow them to do things like steals session cookies, creates false request, create false fields on a page to collect credentials or steal private information. The Score for both of them based on the CVSS can be seen in the following image.



In order to fix the XSS vulnerabilities, the following changes has been made.

CourseResource has been modified in order to fix a vulnerability from the Server Side, the impact of a security breach can be very high . The Title and the Description of the enrolment has been encoded if any script is inserted there it will be modified; this won't allow any user to insert any Persistent XSS attacks.

```
103         String encodeTitle = Encode.forHtmlContent(course.getTitle());
104         course.setTitle(encodeTitle);
105         String encodeDescription= Encode.forHtmlContent(course.getDescription());
106         course.setDescription(encodeDescription);
107         Course result = courseRepository.save(course);
108         return ResponseEntity.ok()
```

The result of the encoding can be seen in the following image.

9	<script>Before Encoding</script>	Before Encoding
10	<script>After Encoding</script>	After Encoding

In order to prevent any DOM-based cross-site scripting in the Client Side, Course-detail.component.html has been modified. The following images shows how the output method was(Line 15) and how it has been modified(14). It's always a bad idea to use a user-controlled input in dangerous sinks.

```
13          <dd>
14              <span>{{ course.description }} </span>
15              <!-- <span [innerHTML] = "course.description | safe: 'html'"></span> -->
16          </dd>
```

3. Identifying and fixing CSRF vulnerability(ies)

As can be seen in the following image from PersistentTokenRememberMeServices inside the Security folder. The token generated when the user log in is stored in the cookies. Those tokens should never be stored there, instead they should be stored into a hidden field on subsequent form submission. In order to fix that issue, the Double-submit cookie pattern can be used. The Double-submit cookie pattern can be stateless on the Server Side, it does not need to store the token between requests. It sets the cookie in response and then on further request we can verify that the token value stored in a cookie is also contains in the request. This way, the attacker is not able to read the tokens.

```
private PersistentToken getPersistentToken(String[] cookieTokens) {
    if (cookieTokens.length != 2) {
        throw new InvalidCookieException("Cookie token did not contain " + 2 +
            " tokens, but contained '" + Arrays.asList(cookieTokens) + "'");
    }
    String presentedSeries = cookieTokens[0];
    String presentedToken = cookieTokens[1];
    Optional<PersistentToken> optionalToken = persistentTokenRepository.findById(presentedSeries);
    if (!optionalToken.isPresent()) {
        // No series match, so we can't authenticate using this cookie
        throw new RememberMeAuthenticationException("No persistent token found for series id: " + presentedSeries);
    }
    PersistentToken token = optionalToken.get();
    // We have a match for this user/series combination
    log.info("presentedToken={} / tokenValue={}", presentedToken, token.getTokenValue());
    if (!presentedToken.equals(token.getTokenValue())) {
        // Token doesn't match series value. Delete this session and throw an exception.
        persistentTokenRepository.delete(token);
        throw new CookieTheftException("Invalid remember-me token (Series/token) mismatch. Implies previous " +
            "cookie theft attack.");
    }

    if (token.getTokenDate().plusDays(TOKEN_VALIDITY_DAYS).isBefore(LocalDate.now())) {
        persistentTokenRepository.delete(token);
        throw new RememberMeAuthenticationException("Remember-me login has expired");
    }
    return token;
}

private void addCookie(PersistentToken token, HttpServletRequest request, HttpServletResponse response) {
    setCookie(
        new String[]{token.getSeries(), token.getTokenValue()},
        TOKEN_VALIDITY_SECONDS, request, response);
}
```

4. Identifying the linkage between vulnerabilities

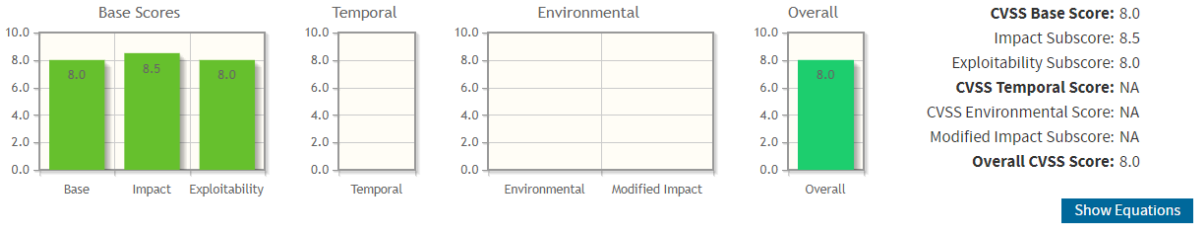
The link between the CSRF and XSS vulnerabilities is if anyone can make a CSRF attack and go inside the system with the Admin's privileges. With those privileges any XSS can be made in case that XSS attacks are not also prevented in parts of the website where just the admin can access. That's why every single part of the website must be prevented against XSS attacks.

5. Identifying and fixing SQL Injection Vulnerability(ies)

The two SQL injection vulnerabilities can be found in the search boxes, one in Courses and the other one in Enrolments.

Both for the same reason, the SQL queries are not well implemented , they were both implemented a normal query and just adding it to it the string passed by the user from the text box. This is a very serious issue that can allow the attacker read and modify sensitive information or execute administration operation in the database.

As mentioned before, they very series issues and it can be seen in the Score obtained bases on the CVSS that can be seen in the following image.



To fix the vulnerability with the *Course* query, parametrisation has been used. The prepared Statements from Java build a dynamic SQL statement safely by binding untrusted data into placeholders within the query. Prepared statement separates queries data from query structure when executing the queries and in case any 'incorrect' parameter is introduced its return an error.

```
public List<Course> searchCourses(@RequestParam String query) {
    log.debug("REST request to search Courses for query {}", query);

    String queryStatement = "SELECT * FROM course WHERE description like ?";
    //String queryStatement = "SELECT * FROM course WHERE description like = % ? %";
    log.info("Final SQL query {}", queryStatement);

    ResultSet rs = null;

    // TODO: use Hibernate language instead
    try (Connection con = DriverManager.getConnection( url: "jdbc:mysql://localhost:3306/NapierUniPortal",  user: "root",  password: "root");
        PreparedStatement stmt = con.prepareStatement(queryStatement)) {
        stmt.setString( parameterIndex: 1, x: "%" + query + "%");

        rs = stmt.executeQuery();
        return extractCourse(rs);
```

For fixing the vulnerability with the *Enrollment* query, which was made with Object Relational Mapping, the same strategy has been used. Each query is prepared before executing it is prepared.

```
@GetMapping("/_search/enrollments")
@Timed
@PreAuthorize("hasRole('ADMIN')")
public List<Enrollment> searchEnrollments(HttpServletRequest request, @RequestParam String query) {
    log.debug("REST request to search Enrollments for query {}", query);
    String comment = "%" + query + "%";
    Session session = HibernateUtil.getSession();
    Query q = session.createQuery( queryString: "select enrollment from Enrollment enrollment where enrollment.comments like :comment");
    q.setParameter( name: "comment", comment);
    return q.list();
}
```

6. Implementing an attack detection mechanism

In order to implement Attack Detection Mechanism, different important events must be monitored. Events like creation of new admin account, any account which has suffered changes on its privileges to admin, failed logins or any JavaScript code which has been added as input. Those types of events may indicate that someone is trying to test vulnerabilities in our system, along with them, is important to obtain the IP address from where the events comes from and store the time so its possible calculate how frequent is happening in order to block that IPAddress from having access to our web portal.

When implementing the attack detection mechanism, I've found errors implementing them. It should be implemented in the *CourseSearchBox* and in the *EnrollmentSearchBox*. In order to implementing it, the OWASP *AppSensor* Framework was used following the way is implemented in the Practical 6 (*SQLInjection*). For implementing two different classes to the Project has been added:

- *EventEmitter*

```
import java.security.GeneralSecurityException;

public class EventEmitter {
    private User bob = new User( username: "enri", new IPAddress( address: "10.10.10.1", new GeoLocation( latitude: 37.596758,  longitude: -121.647992)));
    private DetectionSystem detectionSystem = new DetectionSystem( detectionSystemId: "myclientapp");
    private Gson gson = new Gson();

    private DetectionPoint detectionPoint;

    public EventEmitter(DetectionPoint detectionPoint) { this.detectionPoint = detectionPoint; }

    public ResponseEntity<String> send() {
        System.err.format("Sending event type '%s' from user '%s' and system '%s'%s", detectionPoint.getLabel(), bob.getUsername(), detectionSystem.getDetec
        Event event = new Event(bob, detectionPoint, detectionSystem);
        System.err.println("sending || " + gson.toJson(event) + " ||");

        ResponseEntity<String> responseEntity = null;

        try {
            responseEntity = HttpClient.send(bob, detectionPoint, detectionSystem);
        } catch (GeneralSecurityException e) {
            e.printStackTrace();
        }
        return responseEntity;
    }
}
```

- *HttpClient*

```
public class HttpClient {
    final static String appSensorUrl = "http://localhost:8085/api/v1.0/events";

    public static ResponseEntity<String> send(User user, DetectionPoint detectionPoint, DetectionSystem detectionSystem) throws GeneralSecurityException {
        HttpComponentsClientHttpRequestFactory requestFactory  = new HttpComponentsClientHttpRequestFactory();
        final CloseableHttpClient httpClient = createAcceptSelfSignedCertificateAndAnyHostClient();

        requestFactory.setHttpClient(httpClient);
        Event event = new Event(user, detectionPoint, detectionSystem);
        HttpHeaders headers = new HttpHeaders();
        headers.add( headerName: "X-Appsensor-Client-Application-Name2",  headerValue: "myclientapp");
        HttpEntity<Event> request = new HttpEntity<>(event, headers);
        ResponseEntity<String> response = new RestTemplate(requestFactory).postForEntity(appSensorUrl, request, String.class);
        return response;
    }

    private static CloseableHttpClient createAcceptSelfSignedCertificateAndAnyHostClient() throws KeyManagementException, NoSuchAlgorithmException, KeyStoreException {
        SSLContext sslContext = SSLContextBuilder
                .create()
                .loadTrustMaterial(new TrustSelfSignedStrategy())
                .build();

        HostnameVerifier allowAllHosts = new NoopHostnameVerifier();

        // create an SSL Socket Factory to use the SSLContext with the trust self signed certificate strategy
        // and allow all hosts verifier.
        SSLConnectionSocketFactory connectionFactory = new SSLConnectionSocketFactory(sslContext, allowAllHosts);

        // finally create the HttpClient using HttpClient factory methods and assign the ssl socket factory
        return HttpClients
                .custom()
                .setSSLSocketFactory(connectionFactory)
                .build();
    }

}
```

The detection point was added into the *searchCourses* class of the Coursework project and the also in the xml configuration file into the *AppSensor* project as can be seen in the following to screenshots.

```java
@GetMapping("/_search/courses")
@Timed
public List<Course> searchCourses(@RequestParam String query) {
    log.debug("REST request to search Courses for query {}", query);

    if(query.contains("1=1")){

        DetectionPoint ciel = new DetectionPoint(DetectionPoint.Category.INPUT_VALIDATION,  label: "CIE1");
        EventEmitter cielEmitter = new EventEmitter(ciel);
        cielEmitter.send();
    }

    String queryStatement = "SELECT * FROM course WHERE description like ?";
    //String queryStatement = "SELECT * FROM course WHERE description like = % ? %";


    log.info("Final SQL query {}", queryStatement);

    ResultSet rs = null;

    // TODO: use Hibernate language instead
```

```xml
    <detection-point>
        <category>Input Validation</category>
        <id>CIE1</id>
        <threshold>
            <count>5</count>
            <interval unit="seconds">30</interval>
        </threshold>
        <responses>
            <response>
                <action>log</action>
                <interval unit="minutes">10</interval>
            </response>
            <response>
                <action>logout</action>
            </response>
        </responses>
    </detection-point>
</detection-points>
```

AppSensor included inside the Secure-Software-Development-Module-Coursework folder has been used for it. For running the AppSensor the following code has been used:

cd ~/Secure-Software-Development-Module-Coursework/appsensor/sample-apps/appsensor-ws-rest-server-with-websocket-boot

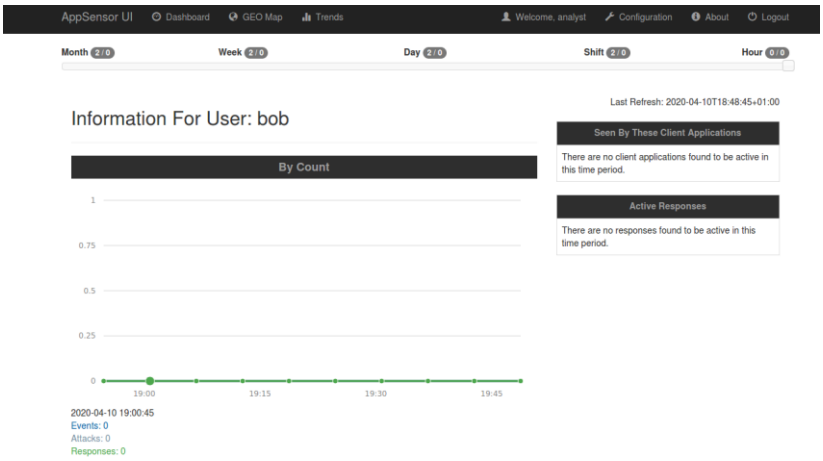
mvn spring-boot:run -DAPPSENSOR_WEB_SOCKET_HOST_URL=ws://localhost:8085/dashboard

For running the AppSensor UI I the following code has been used:

cd ~/Secure-Software-Development-Module-Coursework/appsensor/appsensor-ui

mvn spring-boot:run -DAPPSENSOR_REST_REPORTING_ENGINE_URL=http://localhost:8085 -DAPPSENSOR_CLIENT_APPLICATION_ID_HEADER_NAME=X-Appsensor-Client-Application-Name2 -DAPPSENSOR_CLIENT_APPLICATION_ID_HEADER_VALUE=myclientapp -DAPPSENSOR_WEB_SOCKET_HOST_URL=ws://localhost:8085/dashboard -Dspring.datasource.url=jdbc:mysql://localhost/appsensor –Dspring.datasource.username=appsensor_user -Dspring.datasource.password=appsensor_pass

The AppSensor runs well, and it could be opened but does not detect anything happening in the WebApp.



When testing the SQL vulnerability, the following error has been found on the log.

```
):59:18.075  WARN 4591 --- [ XNIO-2 task-19] .m.m.a.ExceptionHandlerExceptionResolver : Resolved exception caused by Handler execution: org.springframework.web.client.HttpClientErrorException: 400 Bad Request
```

Everything has been checked, nothing has been found the 400 Bad Request appears. Apparently, more people are getting the same error. Everything looks as it should be, but it does not work. The number of detection points should be increased adding the different possible SQL attacks but apart from that the rest of the of the implementation has been implemented.

7. Providing your thoughts

The development of the Web Application has not been done well from a Security Perspective. The team has been working hard in order to fix as many vulnerabilities founded possible.

The system had numerous vulnerabilities regarding Access control. Simple users had access to information to which only the administrator should have access and the privacy of the users was in danger. For this, the team has managed to improve Access control and the privacy of data of our users is not in danger.

In addition to them, two different XSs vulnerabilities have been found, which could pose quite a danger to the system since it allowed users to enter JavaScript code into the database. Both vulnerabilities have been fixed and the system can be said to be free of XSS attacks.

A CSRF vulnerability has also been found, the login token is saved in cookies. The development team has been proposed to implement the Double Cookies system which we hope can be implemented soon.

In addition to the ones mentioned above, the team also found two SQL injection vulnerabilities. These two vulnerabilities were very dangerous because attackers could have access to certain information in the database and could obtain private information from our users. The security team has managed to solve these attacks using the PreparedStatements, which disable any possibility of SQL injection.

Although quite a few of the vulnerabilities have been fixed, the team of engineers has been unable to implement an effective attack detection system. More than half of the implementation has been developed but could not be completed. The engineering team believes that it may be due to the lack of knowledge of something that happens in the WebApplication, but surely with the information provided, this can be corrected by the team of developers.

Among the recommendations to consider for future development, the development team recommends thinking about security before starting to develop the application. In this way it is much easier to develop a

secure application because the development would be done with security in mind as well. If the development is done without thinking about the security issue, it may be the case that certain vulnerabilities are impossible to correct in the future and the system is fragile and accessible.

Part 2

Microsoft DSL - Security Development Lifecycle

The security development life cycle (SDL) is a security control process geared towards software development. The Microsoft SDL process is based on three basic concepts: training, continuous process improvement, and accountability. Continuous training of technical roles within a software development group is essential. By investing appropriately in knowledge transfer, organizations will be able to respond appropriately to changes in technologies and threats. Since security risks are not static, SDL especially stresses the importance of understanding the cause and effect of security vulnerabilities and requires regular evaluation of SDL processes and the introduction of changes in response to technological advances or the new threats. Data is collected to assess training effectiveness, process metrics are used to document compliance, and post-launch metrics help define future changes. Lastly, SDL requires the archiving of all the data necessary to maintain an application in case problems arise. When combined with detailed communication and security response plans, organizations will be able to concisely and forcefully guide all parties involved.

The methodology can be divided into five steps:

- Training. All members of a software development team must receive appropriate training in order to keep abreast of the basics and latest trends in the field of security and privacy. People with technical roles who are directly involved in the development of software programs must attend a security training class at least once a year in fundamental concepts such as: principle of minimum privileges, risk models , buffer overruns, SQL injection, weak cryptography, risk assessment and privacy development procedures.

- Requirements. In the requirements phase, the production team is assisted by a security consultant to review the plans, make recommendations to meet security goals according to project size, complexity, and risk. The production team identifies how security will be integrated into the development process, identifies key security objectives, how the software will be integrated as a whole and will verify that no requirement goes unnoticed.

- Design. The requirements and structure of the software are identified. A secure architecture and design guidelines are defined that identify the critical components for security applying the approach of least privileges and the reduction of the area of attacks. During design, the production team conducts threat modelling at a component-by-component level, detecting the assets that are managed by the software and the interfaces through which those assets can be accessed. The modelling process identifies the threats that could potentially cause damage to an asset and establishes the probability of occurrence and measures are established to mitigate the risk.

- Implementation. During the implementation phase, the software is encoded, tested, and integrated. Threat modelling results guide developers to generate code that mitigates high-priority threats. The coding is governed by standards, to avoid the injection of flaws that lead to security vulnerabilities. The tests focus on detecting vulnerabilities, and fuzz testing techniques and static code analysis are applied using scanning tools developed by Microsoft. Before proceeding to the next phase, a code review is carried out in search of possible vulnerabilities not detected by the scanning tools.

- Verification. In this part, the software is fully functional and is in beta testing phase. Security is subject to a "security push"; which refers to a more thorough review of the code and to running tests on part of the software that has been identified as part of the attack surface.

- Release. At release, the software is subject to a final security review, during a period of two months six prior to delivery in order to know the level of security of the product and the probability of suffering attacks, they are already released to the customer. In case of finding vulnerabilities that put security at risk, it is returned to the previous phase to correct these failures.

Microsoft used the SDL process for the first time to make Windows Vista, significantly reducing vulnerabilities compared to Windows XP SP2.

In order to meet the requirements that any Agile Project requires, Microsoft DSL divides the projects into three distinguished groups depending on the frequency of completion. Every Sprint category contains SDL requirements that are super essential for software security, and without which the software should never be released. The duration of the springs can vary, but all SDL requirements must be completed at the end of these, otherwise the software cannot be released.

The second group, called Bucket Category, comprises tasks that are not as critical as those mentioned above, but that must be carried out regularly during the project. This category can be subdivided into three subgroups that are related tasks. Of the tasks in these groups, the development team only needs to complete one SDL requirements per group. Even so, none of the requirements can be completely ignored since each one of them has been identified to prevent any vulnerability or security problem.

The last group, also called 'One-Time Requirements' contains tasks that only appear once per project and do not need to be repeated once completed. Although these requirements are normally easy to achieve, they are many, and because the developer teams have other tasks to complete, both from the 'every sprint' and the Bucket Category, they are not usually as high a priority, although all of them have a period of completion. that will depend on the importance of these in the project.

The implementation of DSL obviously requires a cost, not only economic, but also time and effort, because all the people involved in the development of the project must be in constant training, this is one of the drawbacks on this implementation. But logically, the expense for this type of development can be always less than the expense that can entail all the reviews, development and testing of each of the security updates that must be done in the future if not a DSL implementation is followed. But the most important thing is, if you get it right up front, the cost of fixing bugs later diminishes rapidly.

References

HOWARD, Michael; LIPNER, Steve. *The security development lifecycle*. Redmond: Microsoft Press, 2006.

Microsoft Security Development Lifecycle(SDL) – version 5.2, https://docs.microsoft.com/en-us/previous-versions/windows/desktop/cc307748(v=msdn.10)