National Conference on Engineering Innovations in Emerging Technologies
In association with International Journal of Scientific Research in Science, Engineering and Technology
Print ISSN: 2395-1990 | Online ISSN : 2394-4099 (www.ijsrset.com)

Security Checker to Detecting Vulnerabilities in Common Weaknesses Enumeration (CWE) Through JAVA Code

Ms.Deepa[1], Dr. Chandramouli H[1], Dr. Anitha N[1]
[1]Department of ISE, East Point College of Engineering and Technology, Bangalore, Karnataka, India

ABSTRACT

In CWE (Common Weakness Enumeration) some weaknesses are categorized that usually occur in any software which are made by the programmers while writing the code so unknowingly done known mistakes are also made but these mistakes only gives a clear path for the attacker to make its way to enter into the code and modify it so as to cause problems that can really do a disastrous approach towards the users.
This paper deals with detecting and finding out those weaknesses which can harm Software application by using a tool called SECCHECK. This Tool has been developed and it will detect few new Software weaknesses. The proposed tool takes Java source files as input and stores each line of input in memory. Then it scans each line of input based on factors that causes vulnerabilities. If it identifies any vulnerability then it displays messages alerting developer to correct these and also calculate the Degree of Insecurity in order to understand the level of insecurity in the application after been tested.
Keywords : - Common Weakness Enumeration (CWE), SECCHECK, Vulnerability, Degree of InSecurity Matric (ISM)

I. INTRODUCTION

Software security is all about Building secure software without flaws. Security is required to provide authentication, integrity, availability and confidentiality. Software security deals with protecting software against malicious attack and other risks by unintended users or by hackers, so that the software continues to work correctly and properly under such risks and threats.
Security aspects have to be taken care during design and implementation phase as unintentional mistakes during coding by the programmer may make the software vulnerable. Poor software design and engineering are the root causes of most security vulnerabilities in deployed systems today. The security of software is threatened at various points throughout its life cycle. The software's security can be threatened during its development, during its deployment, during its operation and during sustainment. CWE is a community-developed list of common software security weaknesses. It serves as a common language, a measuring stick for software security tools, and as a baseline for weakness identification, mitigation, and prevention efforts.

II. LITERATURE SURVEY

Software security is about building secure software: designing software to be secure, making sure that

software is secure, and educating software developers, architects, and users about how to build secure applications. Developing robust, enterprise-level applications are a difficult task, and making them completely secure is virtually impossible. Too often software development organizations place functionality, schedules, and costs at the forefront of their concerns, and make security and quality an afterthought. Nearly all attacks on software applications have one fundamental cause: the code is not secure due to defects in its design, implementation, testing, and operations.

Software security is first and foremost about identifying and managing risks. One of the most effective ways to identify and manage risk for an application is to iteratively review its code throughout the development cycle. Application security is the key risk area for exploits, and exploits of applications can be devastating [4].

Vulnerability is an error that an attacker can exploit. Many types of vulnerabilities exist in software systems, including local implementation errors, inter-procedurally interface errors (such as a race condition between an access control check and a file operation), design-level mistakes (such as error handling and recovery systems that fail in an insecure fashion), and object-sharing systems that mistakenly include transitive trust issues.

Vulnerabilities typically fall into two categories:

1. Bugs at the implementation level and
2. Flaws at the design level.

2.1 Detection of Vulnerabilities in Design Phase

The Design Phase seeks to develop detailed specifications that emphasize the physical solution to the user's information technology needs. The system requirements and logical description of the entities, relationships, and attributes of the data that were documented during the Requirements Analysis Phase are further refined and allocated into system and database design specifications that are organized in a way suitable for implementation within the constraints of a physical environment (e.g., computer, database, facilities).

The cost associated with addressing software problems increases as the lifecycle of a project matures. The cost of finding and fixing a bug after a software product has been released can be 100 times more expensive than solving the problem in the requirements or design phase. Thus, patching vulnerable software after release can be a costly way of securing applications. Furthermore, patches are not always applied by owners/users of the vulnerable software; patches can contain yet more vulnerabilities [24].

Since design phase prepares skeleton of the software, making changes and corrections in the phase is much easier than to make them in the subsequent phases. A single design flaw may manifest itself causing serious security related consequences. The very architecture of the application should take security into account from the outset, and that concern should be followed through down to implementation and deployment [5].

The cost of defects, especially security defects, is (or can be) a lot higher once the application is deployed than before deployment – defects are usually especially cheap if caught at early design phases.

In order to minimize vulnerabilities, the propagation of vulnerabilities must be controlled. Further, the means by which these propagations are made (i.e. the design characteristics), need to be analysed to make appropriate decision regarding the same [25].

2.1.1 Vulnerability Analysis

A security assessment or security vulnerability analysis is a subset of a process called enterprise risk management, which involves evaluating and prioritizing all risks to an organization, security being one of them. For instance, from an enterprise risk management perspective, the security risk could be vulnerability to assets, people, business, brand and reputation. To examine this risk, a security vulnerability analysis would evaluate an organization

to identify, validate and prioritize vulnerabilities that could produce a security incident. This incident could be as mundane as product loss or as catastrophic as a shooting in a facility.

A security vulnerability analysis seeks out root causes for security vulnerability and applies physical, technical and operational controls to deter, delay and minimize the impact on the organization for an incidence [7]. The security vulnerability analysis validates vulnerabilities to upper management and helps procure money for improvements. These improvements could be establishing a security program, purchasing technology, performing upgrades to lighting or physical security, training, improving awareness, and so on. Vulnerability analysis should be done in all of the development phase since they might be injected in many phases such as requirement phase, design level mistakes, implementation errors etc., as shown in the Figure 2.1 [26].

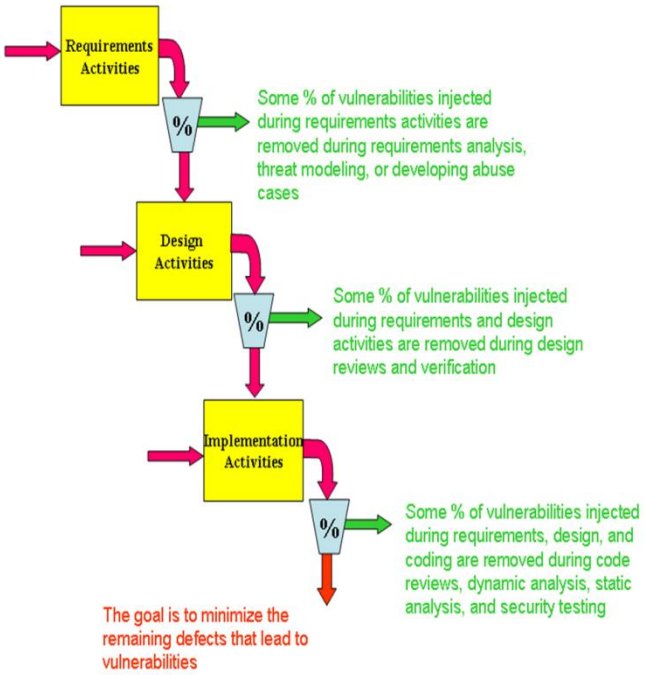

Fig 2.1: Injection of Vulnerabilities in Different Phases

2.1.2 Vulnerability Modeling

Most of the vulnerabilities presented in the previous section could be prevented if the software is developed more carefully, avoiding the introduction of vulnerabilities that could be exploited by attackers. One solution is in the improvement of the knowledge and understanding of software developers about: known vulnerabilities, causes, threats, attacks and counter measures. Models are in fact adequate to implement such solution. There is for instance a vulnerability model called Vulnerability Cause Graph (VCG) which "is a directed acyclic graph that contains one exit node representing the vulnerability being modeled, and any number of cause nodes, each of which represents a condition or event during software development that might contribute to the presence of the modeled vulnerability".

An example of a VCG representing a known buffer overflow in xpedx (CVE-2005-3192) taken from [2] in the above figure. In this graph we can observe the different causes and possible scenarios or sequence actions that could lead to the introduction of this kind of vulnerability. The VCG is helpful to understand what can cause the vulnerability. If causes are well understood, then they could be avoided in the development process.

2.1.3 Software Inspection

The software inspection process consists in reading or visually inspecting the program code or documents in order to find any defects and correct them early in the development process. When the defect is found soon the less expensive it becomes to fix. However, a good inspection depends then on the ability and expertise of the inspector, and the kind of defects he is looking for. Usually during the software inspection, it is necessary to look for any possible defects during the security inspections. In the following sections we introduce two inspection methods that intend to codify the implicit knowledge of security experts regarding how to check for correct implementation of security goals and how to search for vulnerabilities [28].

III. ARCHITECTURE

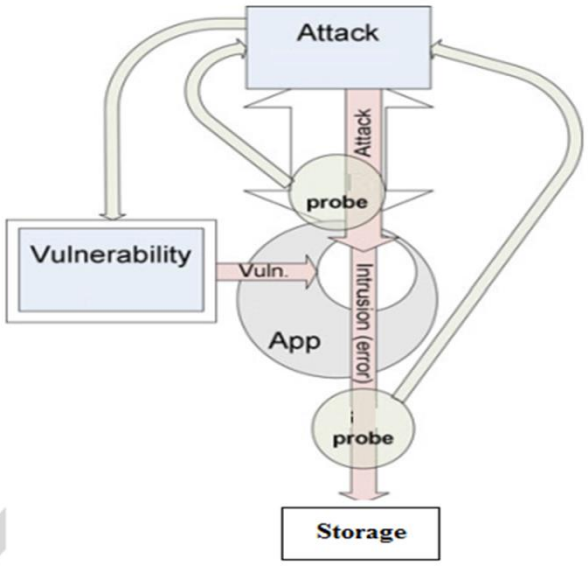

Fig 3.1: Architecture of the Proposed System

In this system architecture, analyses the java source code in the application and check for the vulnerability attacks in the source code and notify them.

Load the attackers in source code of java files.

Application has to identify the type of vulnerabilities in the source code and measure the degree of insecurity metric as well.

IV. PROPOSED METHOD

We implement following security ids from CWE.

1. **CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')**
2. **CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')**
3. **CWE-94: Improper Control of Generation of Code ('Code Injection')**
4. **CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer**
5. **CWE-185: Incorrect Regular Expression**
6. **CWE-190: Integer Overflow or Wraparound**
7. **CWE-202: Exposure of Sensitive Data Through Data Queries**
8. **CWE-233: Improper Handling of Parameters Advantages**:

Our solution relies on scanning & identifying the security vulnerabilities rather than executing them. We identify the line number in which vulnerability is present & it helps the developer to fix it easily.

V. IMPLEMENTATION

Generic Steps and its logic behind the working for each vulnerability IDS

Step 1.: Load the java files
Step 2.: Apply for loop
Step 3.: go to each file
Step 4.: read the each line in the file
Step 5.: Apply condition
Step 6.: check the Patterns
Step 7.: if patterns correct
Step 8.: Vulnerability found
Step 9.: end if
Step 10.: else
Step 11.: No vulnerability found
Step 12.: END

5.1. PseudoCode in Java for CWE IDs

CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')

```
public String coordinateTransformLatLonToUTM(String coordinates)
{
String utmCoords = null;
try {
String latlonCoords = coordinates;
Runtime rt = Runtime.getRuntime();
Process exec = rt.exec("cmd.exe /C latlon2utm.exe -" + latlonCoords);
// process results of coordinate transform

// ...
}
catch(Exception e) {...}
return utmCoords;
}
```

CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

```
string userName = ctx.getAuthenticatedUserName();
string query = "SELECT * FROM items WHERE owner = '" + userName +
AND itemname = '" + ItemName.Text + "'";
sda = new SqlDataAdapter(query, conn);
DataTable dt = new DataTable();
sda.Fill(dt);
```

CWE-94: Improper Control of Generation of Code ('Code Injection')

```
MessageFile = "cwe-94/messages.out";
if (get["action"] == "NewMessage") {
name = get["name"];
message = get["message"];
handle = fopen(MessageFile, "a+");
fwrite(handle, "<b>name</b> says 'message'<hr>\n");
fclose(handle);
system.out.println( "Message Saved!<p>\n");
}
else if (GET["action"] == "ViewMessages") {
include(MessageFile);
}
```

CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer

```
void host_lookup(char *user_supplied_addr){
struct hostent *hp;
in_addr_t *addr;
char hostname[64];
in_addr_t inet_addr(const char *cp);

/*routine that ensures user_supplied_addr is in the right format for conversion */

validate_addr_form(user_supplied_addr);
addr = inet_addr(user_supplied_addr);
hp = gethostbyaddr( addr, sizeof(struct in_addr), AF_INET);
strcpy(hostname, hp->h_name);
}
```

CWE-185: Incorrect Regular Expression

```
phone = GetPhoneNumber();
if (phone =~ /\d+-\d+/) {
// looks like it only has hyphens and digits
system.out.println("lookup-phone $phone");
}
else {
system.out.println("malformed number!");
}
```

CWE-190: Integer Overflow or Wraparound

```
float calculateRevenueForQuarter(long quarterSold) {...}

int determineFirstQuarterRevenue() {
...
// Calculate quarterly total
long quarterSold = JanSold + FebSold + MarSold;

// Calculate the total revenue for the quarter
quarterRevenue = calculateRevenueForQuarter(quarterSold);

...
}
```

CWE-233: Improper Handling of Parameters

```
IntentFilter filter = new IntentFilter("com.example.URLHandler.openURL");
MyReceiver receiver = new MyReceiver();
registerReceiver(receiver, filter);
...

public class UrlHandlerReceiver extends BroadcastReceiver {
@Override
public void onReceive(Context context, Intent intent) {
if("com.example.URLHandler.openURL".equals(intent.getAction())) {
String URL = intent.getStringExtra("URLToOpen");
int length = URL.length();

...
}
}
}
```

5.2. Weakness Identification in Ids

Weaknesses that are rarely exploited will not receive a high score, regardless of the typical severity associated with any exploitation. This makes sense, since if developers are not making a particular mistake, then the weakness should not be highlighted in the CWE.

Weaknesses with a low impact will not receive a high score. This again makes sense, since the inability to cause significant harm by exploiting a weakness means that weakness should be ranked below those that can.

Weaknesses that are both common and can cause harm should receive a high score.

To check the Score with proper parameters:

If the total number of errors identified in the tested java file is more than ISM score values gets increased

If the total number of errors identified in the tested java file is less then ISM score value gets decreased

VI. EXPERIMENTAL RESULTS

To Calculate ISM

The level of danger presented by a particular CWE is then determined by multiplying the severity score by the frequency score.

Score (CWE_X) = Fr (CWE_X) * Sv (CWE_X) * 100

There are a few properties of the scoring method that merit further explanation.

VII.CONCLUSION

Vulnerability is a flaw within the source code which can be exploited by attacker to hack the code. Vulnerability is a weak spot in the software, which if exploited will result in the compromise of the system. Vulnerability is the intersection of three elements: a system susceptibility or flaw, attacker access to the flaw, and attacker capability to exploit the flaw. To exploit vulnerability, an attacker must have at least one applicable tool or technique that can connect to a system weakness. In this frame, vulnerability is also known as the attack surface.

Existence of vulnerabilities implies weaker software. In order to make software weakness free, vulnerabilities must be detected and corrected. Vulnerabilities must be identified during development phase of the software to produce better software product. The proposed tool finds ten vulnerabilities in Java source code.

The vulnerabilities identified are as follows

- **CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')**
- **CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')**
- **CWE-94: Improper Control of Generation of Code ('Code Injection')**
- **CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer**
- **CWE-185: Incorrect Regular Expression**
- **CWE-190: Integer Overflow or Wraparound**
- **CWE-202: Exposure of Sensitive Data Through Data Queries**
- **CWE-233: Improper Handling of Parameters**

The proposed tool takes Java source files as input and stores each line of input in memory. Then it scans each line of input based on factors that cause vulnerabilities. If it identifies any vulnerability then it displays a warning message and shows the degree of insecurity.

The tool can be further enhanced to find other software vulnerabilities that cause unsafe software.

VIII. ACKNOWLEDGEMENT

The project is funded by VGST, Government of Karnataka under K-FIST L1, with GRD No: 264 and the authors would like to thank the VGST Group for providing this opportunity.

IX. REFERENCES

[1]. www.ece.cmu.edu/~dbrumley/courses/18732f09/
[2]. https://buildsecurityin.us-cert.gov/bsi/547.html#dsy547-BSI_princ
[3]. http://en.wikipedia.org/wiki/Vulnerability_(computing)
[4]. Asoke K.Talukder, Manish Chaitanya."Architecting Secure Software Systems", 2009
[5]. http://rlc.vlinder.ca/blog/2009/09/security-at-the-design-phase-examples-review/
[6]. http://msdn.microsoft.com/en-us/library/windows/desktop/cc307414.aspx
[7]. Steven Lavenhar."Code Analysis", 2008.
[8]. Robert C. Seacord Allen D. Householder." A Structured Approach to Classifying Security Vulnerabilities", January 2005
[9]. CLASP Vulnerability View — Classes in CLASP Taxonomy, March 2006
[10]. http://makingsecuritymeasurable.mitre.org/docs/cwe-intro-handout.pdf
[11]. http://msdn.microsoft.com/en-us/library/windows/desktop/cc307416.aspx
[12]. Michal Chmielewski, Neill Clift, Sergiusz Fonrobert and Tomasz Ostwald ,"Find and Fix Vulnerabilities Before Your Application Ships".
[13]. Steven M. Christey, Janis E. Kenderdine, John M.Mazella and Brendan Miles. "CWE V2.0": 2011
[14]. http://en.wikipedia.org/wiki/Off-by-one_error
[15]. http://cwe.mitre.org/data/definitions/789.html
[16]. http://cwe.mitre.org/data/definitions/20.html.
[17]. http://cwe.mitre.org/data/definitions/754.html.
[18]. http://en.wikipedia.org/wiki/Arithmetic_underflow
[19]. http://my.safaribooksonline.com/book/software-engineering-and-development/0321166078/floating-point-arithmetic/ch08lev1sec4
[20]. http://javapapers.com/core-java/java-overflow-and-underflow/
[21]. Dead Code: http://en.wikipedia.org/wiki/Dead_code
[22]. http://link.springer.com/static-content/lookinside/891/chp%253A10.1007%252F978-3-642-35606-3_16/000.png
[23]. collaboration.csc.ncsu.edu/laurie/Papers/ICSE_Final_MCG_LW.pdf
[24]. seij.dce.edu/vol-2/paper5.pdf
[25]. https://buildsecurityin.us-cert.gov/articles/knowledge/sdlc-process/secure-software-development-life-cycle-processes
[26]. https://lh6.googleusercontent.com/-S-VcaHPug00/UVHExiEmXAI/AAAAAAAALQs/Jik0EqgAvjs/s1867/2013%25252006%25253A25.jpg
[27]. www-lor.int-evry.fr/~anna/files/sec-mda09.pdf
[28]. http://en.wikipedia.org/wiki/Penetration_test
[29]. http://en.wikipedia.org/wiki/Security_testing
[30]. finalize() Method Declared Public: http://cwe.mitre.org/data/definitions/583.html
[31]. https://www.securecoding.cert.org/.../MET12-J.+Do+not+use+finalizers
[32]. Improper Initialization: http://cwe.mitre.org/data/definitions/665

[33]. http://books.google.co.in/books?id=8d-qU8K0BN4C&pg=PT128&lpg=PT128&dq=improper+initialization&source=bl&ots=TjhUdhx-1G&sig=y3Di4gH_Iea6_BrzdsbFTaheIso&hl=en&sa=X&ei=xm21Ucs4zpOuB9mVgMAN&ved=0CEoQ6AEwBjge
[34]. http://www.oracle.com/technetwork/java/seccodeguide-139067.html#5
[35]. Michal Chmielewski, Neill Clift, Sergiusz Fonrobert and Tomasz Ostwald ,"Find and Fix Vulnerabilities Before Your Application Ships".
[36]. http://www.its.ny.gov/pmmp/guidebook2/SystemImplement.pdf