SecCheck: A Tool for Detection of Vulnerabilities and for Measuring Insecurity in Java Programs

Priyadarshini.R [1], Nivedita Ghosh [2] and Anirban Basu [3]

(1) Department of CSE, East Point College of Engineering & Technology, Bengaluru, (India)
Email: priya87darshini@gmail.com
(2) Department of CSE, East Point College of Engineering & Technology, Bengaluru, (India)
Email: gh.nivedita@gmail.com,
(3) Department of CSE, East Point College of Engineering & Technology, Bengaluru, (India)
Email: abasu@anirbanbasu.in

ABSTRACT

Software vulnerability is a weakness that can be exploited to get access to the code making the software highly insecure. To make the software secure, vulnerabilities must be identified and corrected. As identifying weaknesses manually in large programs is time consuming, the process needs to be automated. This paper discusses a tool called SecCheck developed to identify vulnerabilities in Java code. The tool takes Java source files as input, stores each line in memory and scans to find vulnerabilities. A warning message is displayed when vulnerability is found. The tool can detect critical software vulnerabilities not found by most of the other tools as well as calculate *Degree of Insecurity*, a metric defined in this paper. SecCheck has been used to calculate the Degree of Insecurity in two classes of programs: one written by experienced Java programmers and the other by students. The experimental results are discussed.

Keywords: Common Weakness Enumeration, Degree of Insecurity, SecCheck Tool, Security Threat, Software Vulnerability, Tools for Vulnerability Detection.

1- INTRODUCTION

The security of software is threatened at various stages throughout its lifecycle, inadvertently due to mistakes committed by developers or by intentional hacking. Poor software design and implementation are the primary causes of most security weaknesses [1][2].Security threats may also come from web sites and web applications (webapps). Data centres and other assets used for hosting web sites and their associated systems need to be protected from all types of threats.

Annual Surveys conducted by the Computer Security Institute, the FBI, PwC, Forbes etc. reveal that cyber criminals are implementing increasingly sophisticated methods for targeting specific computer systems and organizations: big and small. A recent vulnerability assessment study performed on more than 250 Web applications from e-commerce, online banking, enterprise collaboration, and supply chain management sites found out that at least 92% of web applications are vulnerable to some form of hacker attacks. It is doubtful whether the present Information Security techniques will be able to protect critical software systems unless security mechanisms are made an inherent part of the software.

Java has emerged as the language of choice for building large complex Web-based systems, partly because of language safety features that disallow direct memory access and eliminate problems such as buffer overruns. However, despite these features, it is possible to make logical programming errors that lead to vulnerabilities such as SQL injections and cross-site scripting attacks [3]. A simple programming mistake can leave a Web application vulnerable to unauthorized data access, unauthorized updates or deletion of data, and crashing of applications leading to denial-of-service attacks [4].

Due to processing of critical information, it is important that software is protected against malicious attacks and other risks so that it continues to function correctly even under such potential threats. Threats can be identified using application threat modelling and then evaluated by vulnerability assessment. Nowadays, lot of attention is being given on building secure software and on detecting vulnerabilities by static analysis [5][6]. Many types of vulnerabilities exist in software systems injected in design and in implementation phases, such as local implementation errors, inter procedural 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, topology of the web of links, deep linking, unspecified image dimensions), and object-sharing systems [7]. Efforts are required during design and implementation to make the software secure and to protect the software against malicious attacks and other risks.

This paper discusses vulnerabilities that are injected in Java programs during coding phase and describes a tool developed to detect the weaknesses and alert the developer about these. The tool developed by the authors, and named SecCheck detects vulnerabilities in any Java program caused by off-by-one error, uncontrolled memory allocation, improper input validation, improper check for unusual or exceptional conditions, arithmetic underflow, dead code, de-serialization of un-trusted data, incorrect conversion between numeric types, finalize() method declared public, and improper initialization, absolute path traversal, uncontrolled resource consumption, uploading files of dangerous type, manipulating input to file system calls, URL redirection to un-trusted site, client-side enforcement of server- side security, sensitive cookies of https session without ‘secure’ attribute, improper neutralization of http headers for scripting syntax, and information exposure through log files.

The tool described here not only detects the vulnerabilities present in the code but also calculates the *Degree of Insecurity* of input Java program. The effectiveness of the tool has been studied by using it on calculating the *Degree of Insecurity* in two classes of programs: one written by experienced Java Programmers and the other by students.

Section 2 discusses the present state of research in the area and Table 1 presents the features of other tools available for detection of vulnerabilities and provides a comparison of these with SecCheck. Section 3 discusses the consequences of the vulnerabilities generally found in any software and detected by the tool. Section 4 defines *Degree of Insecurity* in a program and Section 5 discusses the working of SecCheck, Section 6 discusses the *Degree of Insecurity* found by using the tool in two classes of programs: one by professionals and another one by students. As expected the Java programs written by the students are more prone to security attacks than those written by professionals.

2- PRESENT STATE OF RESEARCH

Due to increased incidents of theft of information, Software Security is gaining lot of attention [1][2]. CWE [7], a consortium which creates a catalog of software weaknesses and vulnerabilities has been formed. It assists organizations in selecting the right software tools and learning about possible weaknesses and their possible impact and is enabling more effective discussion, description, selection, and use of software security tools and services. The catalogue prepared by CWE is helping the development teams to find weaknesses in source code and in operational systems as well as have better understanding and management of software weaknesses relating to architecture, design and code. However, the current state of preventive and corrective methodologies is inadequate.

Various tools as listed in Table 1 are available to identify and detect the vulnerabilities present in source code in different programming languages. However none of the tools discussed in Table 1 can detect all possible vulnerabilities needed to safeguard against attacks from hackers.

The number of vulnerabilities detected by SecCheck is more compared to the existing tools. Besides, the available tools do not calculate the *Degree of Insecurity*. These features make SecCheck more useful to the software developers.

3- COMMON VULNERABILITIES IN SOFTWARE

Software vulnerabilities that are commonly found in Java programs as discussed in CWE [7] are:

- Off-by-one Error
- Uncontrolled Memory Allocation
- Improper Input validation
- Improper Check For Unusual Or Exceptional Conditions
- Arithmetic Underflow
- Dead Code
- Deserialization of untrusted data
- Incorrect Conversion between Numeric Types
- finalize() Method Declared Public
- Improper Initialization
- Absolute Path Traversal
- Uncontrolled Resource Consumption(‘Resource Exhaustion’)
- Unrestricted Upload of Files with Dangerous Type
- Manipulating Input to File System Calls
- URL Redirection to Untrusted Site(‘Open Redirect’)
- Client-Side Enforcement of Server- Side Security
- Sensitive Cookies of HTTPS Session Without ‘Secure’ Attribute
- Improper Neutralization of HTTP Headers for Scripting Syntax
- Information Exposure through Log Files

These vulnerabilities are detected by the SecCheck tool in any Java program and a warring message is displayed on detection along with line number where it occurs.

The above mentioned vulnerabilities are briefly discussed here along with the consequences of their presence.

3-1 OFF-BY-ONE ERROR

Off-by-one error (OBOE) is a logic error involving the discrete equivalent of a boundary condition. It often occurs in computer programming when an iterative loop iterates one time too many or too few. Usually this problem arises when a programmer fails to take into account that a sequence starts at zero rather than one (as with array indices in many languages), or makes mistakes such as using "is less than or equal to" where "is less than" should have been used in a comparison. This can also occur in a mathematical context [8].

This may result in erratic program behaviour, including memory access errors, incorrect results, a crash, or a breach of system security. Thus, they are the basis of many software vulnerabilities and can be maliciously exploited.

Consequences

a. Overwriting of the least significant bit in the frame pointer of a block of allocated memory can corrupt data, crash the program, or allow the execution of malicious code.

b. Off-by-one error causes an exploitable condition where an attacker can hijack the local variables for the calling routine.

c. Off-by-one error may result in erratic program behaviour, including memory access errors, incorrect results, a crash, or a breach of system security.

3-2 UNCONTROLLED MEMORY ALLOCATION

The product allocates memory based on an untrusted size value, but it does not validate or incorrectly validates the size, allowing arbitrary amounts of memory to be allocated [9].

This vulnerability is possible in Java by initial size parameters in collections. The code accepts an untrusted size value and allocates a buffer to contain a string of given size [9].

Uncontrolled memory allocation leads to crashes of application due to out of memory conditions. Attackers can make use of this to hack the data.

Consequences

a. Uncontrolled memory allocation leads to crash the application.

b. It may also lead to consumption of a large amount of memory on the system.

3-3 IMPROPER INPUT VALIDATION

The product does not validate or incorrectly validates input that can affect the control flow of a program [10].

When software does not validate input properly, an attacker is able to craft the input in a form that is not expected by the rest of the application. This will lead to parts of the system receiving unintended input which may result in altered control flow, arbitrary control of a resource or arbitrary code execution [10].

Consequences

a. An attacker could provide unexpected values and cause a program crash or excessive consumption of resources, such as memory and CPU.

b**.** An attacker could read confidential data if they are able to control resource references.

c**.** An attacker could use malicious input to modify data or possibly alter control flow in unexpected ways, including arbitrary command execution.

3-4 IMPROPER CHECK FOR UNUSUAL OR EXCEPTIONAL CONDITIONS

The software does not check or improperly checks for unusual or exceptional conditions that are not expected to occur frequently during day to day operation of the software [11].

The programmer may assume that certain events or conditions will never occur or do not need to be worried about, such as low memory conditions, lack of access to resources due to restrictive permissions, or misbehaving clients or components. However, attackers may intentionally trigger these unusual conditions, thus violating the programmer's assumptions, possibly introducing instability, incorrect behaviour, or vulnerability [11].

Consequences

a. The data which were produced as a result of a function call could be in a bad state upon return. If the return value is not checked, then this bad data may be used in operations, possibly leading to a crash or other unintended behaviours.

b**.** Leads to resource exhaustion, low memory conditions,

c. An attacker may be able to assert control before the software has fully exited.

3-5 ARITHMETIC UNDERFLOW

The term arithmetic underflow (or "floating point underflow", or just "underflow") is a condition in a computer program where the result of a calculation is a smaller number than the computer can actually store in memory [12].

Arithmetic underflow can occur when the true result of a floating point operation is smaller in magnitude (that is, closer to zero) than the smallest value which can be represented as a normal floating point number in the target data type. Underflow can in part be regarded as negative overflow of the exponent of the floating point value. For example, if the exponent part can represent values from −128 to 127, then a result with absolute value less than 2−127 may cause underflow (assuming that the exponent −128 is reserved for values like −∞ which have no "normal" representation).

Consequences

a. Java runtime does NOT issue an error/warning message but produces an incorrect result [12].

b**.** On the other hand, integer division produces a truncated integer and results in so-called underflow. For example, 1/2 gives 0, instead of 0.5. Again, Java runtime does NOT issue an error/warning message, but produces an imprecise result [12].

3-6 DEAD CODE

In computer programming, dead code is code in the source code of a program which is executed but whose result is never used in any other computation. The execution of dead code wastes computation time as its results are never used [13].

While the result of a dead computation may never be used, the dead code may raise exceptions or affect some global state, thus removal of such code may change the output of the program and introduce unintended bugs [13].

Consequences

a. Occupies unnecessary memory.

b**.** From the perspective of program maintenance; time and effort may be spent maintaining and documenting a piece of code which is in fact unreachable, hence never executed.

3-7 DESERIALIZATION OF UNTRUSTED DATA

The application de-serializes un-trusted data without sufficiently verifying that the resulting data will be valid [14].

It is often convenient to serialize objects for communication or to save them for later use. However, de-serialized data or code can often be modified without using the provided accessor functions if it does not use cryptography to protect itself. Furthermore, any cryptography would still be client-side security -- which is a dangerous security assumption [14].

Data that is un-trusted cannot be trusted to be well-formed.

Consequences

a. An attacker may be able to replace the intended file with a file that contains arbitrary malicious code which will be executed.

b. Code could potentially make the assumption that information in the deserialized object is valid. Functions which make this dangerous assumption could be exploited.

3-8 INCORRECT CONVERSION BETWEEN NUMERIC TYPES

When converting from one data type to another, such as long to integer, data can be omitted or translated in a way that produces unexpected values. If the resulting values are used in a sensitive context, then dangerous behaviours may occur [15].

Consequences

a. The program could wind up using the wrong number and generate incorrect results.

b. Behaviour may have unintended consequences.

c. Leads to integer overflow and heap overflow.

3-9 FINALIZE () METHOD DECLARED PUBLIC

The program violates secure coding principles for mobile code by declaring a finalize () method public [16].

A program should never call finalize explicitly, except to call super.finalize() inside an implementation of finalize(). In mobile code situations, the otherwise error prone practice of manual garbage collection can become a security threat if an attacker can maliciously invoke one of your finalize() methods because it is declared with public access [16].

Consequences

a. Can Alter execution logic

b. Execute unauthorized code or commands

c. Modify application data

3-10 IMPROPER INITIALIZATION

The software does not initialize or incorrectly initializes a resource, which might leave the resource in an unexpected state when it is accessed or used [17].

This can have security implications when the associated resource is expected to have certain properties or values, such as a variable that determines whether a user has been authenticated or not [17].

Consequences

a. Bypass of security may occur.

b**.** The uninitialized data may contain values that cause program flow to change in ways that the programmer did not intend.

c**.** The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.

3-11 ABSOLUTE PATH TRAVERSAL

The attack Absolute Path Traversal takes place when a software uses an external input which is used to construct a pathname that is been created in restricted directory but does not properly neutralize absolute path sequences such as “/abs/path” that can resolve that location that is outside of that directory so by this attacker traverses the file system to access files or directories that are outside of that directory [18]. By such kind of weakness attackers can traverse the file system to access files or directories that are outside of the restricted directory.

Consequences

a. It exploits unchecked user input to control which files are accessed on the server.

b. Attacker can traverse the file system remotely and can change contents of file.

c. The attacker may be able to overwrite, delete, or corrupt unexpected critical files such as programs, libraries, or important data.

3-12 UNCONTROLLED RESOURCE CONSUMPTION (‘RESOURCE EXHAUSTION’)

In this attack the software does not properly restrict the size or amount of resources that are requested or influenced by an actor, which can be used to consume more resources than intended.

Resources include memory, file system storage, CPU. If an attacker can trigger the allocation of these limited resources but the number or size of resources is not controlled, then the attacker could cause a **denial of service** that consumes all available resources.

A Memory Exhaustion Attack against an application could slow down the application as well as its host operating system [19].

Resource Exhaustion problems have at least 2 common causes:

a. Error conditions and other exceptional circumstances
b. Confusion over which part of the program is responsible for releasing resource

Consequences

a**.** Most hackers spoof their IP or bounce data off of another machine so that it is hard to track them.

b. If an attacker can trigger the allocation of resources, but the number or size of the resources is not controlled, then the attacker could cause a **denial of service** that consumes all available resources.

c. Many DoS attacks, such as the *Ping of Death and Teardrop* attacks, exploit limitations in the TCP/IP protocols.

3-13 UNRESTRICTED UPLOAD OF FILE WITH DANGEROUS TYPE

Unrestricted uploading of files with any extension or any content size can be dangerous because it can allow an attacker to upload malicious files and very long length files which can be again virus that can destroy the confidential files in the system. Even the web application running can allow for uploading file of any extension which can indirectly affect the system while accepting the file of any type.

However, the attacker may upload files having virus or malicious file directly or even large byte file so after this system can get into access anywhere and is not safe anymore even so it becomes very difficult to analyse the depth of the attack also [20].

Consequences

a. Arbitrary code execution is possible if an uploaded file is interpreted and executed as code by the recipient.

b. The web server might be used as a warez server by a bad guy in order to be host of malwares, illegal software, steganographic objects, and so on.

c. A malicious file can be uploaded on the server in order to have a chance to be executed by administrator or webmaster later.

3-14 MANIPULATING INPUT TO FILE SYSTEM CALLS

This attack deals with an attacker who manipulates inputs to the target software which the target software passes to file system calls in the OS. The goal is to gain access to, and perhaps modify, areas of the file system that the target software did not intend to be accessible [21].

Consequences

a. In order to create a valid file injection, the attacker needs to know what the underlying OS is.

b. The attacker may steal information or directly manipulate files (delete, copy, flush, etc.)

c. Attacker’s motive is to identify file system entry point and execute against an over privileged system interface.

3-15 URL REDIRECTION TO UNTRUSTED SITE ('OPEN REDIRECT')

A web application accepts a **user-controlled input** that specifies a link to an external site, and uses that link in a Redirect. This simplifies phishing attacks.

An http parameter may contain a **URL value** and could cause the web application to redirect the request to the specified URL. By **modifying the URL value** to a malicious site, an attacker may successfully launch a phishing scam and steal user credentials. Because the server name in the modified link is identical to the original site, phishing attempts have a more trustworthy appearance [22].

Consequences

a. The user may be redirected to an un-trusted page that contains malware which may then compromise the user's machine.

b. The user may be subjected to phishing attacks by being redirected to an un-trusted page.

c. The phishers may then steal the users' credentials and then use these credentials to access the legitimate web site.

3-16 CLIENT-SIDE ENFORCEMENT OF SERVER-SIDE SECURITY

In this above attack the software is composed of a server that relies on the client to implement a mechanism that is intended to protect the server.

When the server relies on protection mechanisms placed on the client side, an attacker can modify the client-side behaviour to bypass the protection mechanisms resulting in potentially unexpected interactions between the client and server [23]. The consequences will vary, depending on what the mechanisms are trying to protect.

Consequences

a. Client-side validation checks can be easily bypassed.

b**.** Client-side checks for authentication can be easily bypassed, allowing clients to escalate their access levels and perform unintended actions.

c. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely.

3-17 SENSITIVE COOKIE IN HTTPS SESSION WITHOUT 'SECURE' ATTRIBUTE

The software is composed of a server that relies on the client to implement a mechanism that is intended to protect the server.

The Secure attribute for sensitive cookies in HTTPS sessions is not set, which could cause the user agent to send those cookies in plaintext over an HTTP session [24].

Consequences

a. If the Secure attribute is not set for sensitive cookies in HTTPS sessions, this could cause the user agent to send those cookies in plaintext over an HTTP session with the product.

b. When the secure flag is not set for the session cookie in an https session, this can cause the cookie to be sent in http requests and make it easier for remote attackers to capture this cookie.

c. If we log in to a site with plain HTTP it is completely insecure, and anyone in a public WLAN can sniff the data.

3-18 IMPROPER NEUTRALIZATION OF HTTP HEADERS FOR SCRIPTING SYNTAX

The application does not neutralize or incorrectly neutralizes web scripting syntax in HTTP headers that can be used by web browser components that can process raw headers, such as Flash.

An attacker may be able to conduct cross-site scripting and other attacks against users who have these components enabled [25].

Consequences

a. There are chances of occurring Cross Site Scripting and Cache Poisoning Attacks.

b. Attackers may be able to obtain sensitive information like user credentials or log file information.

c. If headers are not properly neutralized then there are can also be that attacker can run arbitrary code that can be malicious.

3-19 INFORMATION EXPOSURE THROUGH LOG FILES

Information written to log files can be of a sensitive nature and give valuable guidance to an attacker or expose sensitive user information.

While logging all information may be helpful during development stages, it is important that logging levels be set appropriately before a product ships so that sensitive user data and system information are not accidentally exposed to potential attackers [26].

Consequences

a. Logging sensitive user data often provides attackers with an additional, less-protected path to acquiring the information.

b**.** Malicious attackers will try to get your login information for any of the services you're using, which will then allow them to break into the rest.

c. User information regarding bank credentials or credit card details can get exposed if given in simple text format in log file.

4- DEGREE OF INSECURITY IN A PROGRAM

Each of the weaknesses discussed in this paper has been assigned a severity level defined in CWE. In this paper we define a metric for calculating the Degree of Insecurity (referred to as ISM).

ISM= $\sum_{i=1}^{m} Wi * N i$
where,

ISM stands for the Degree of Insecurity,
i is the type of vulnerability where i=1,2,....m
W_i is the Severity of Vulnerability in the software
Ni is the frequency of occurrence of vulnerability i.

The values of severity is taken from [7] and given below in Table 2

Table 2: Severity of Vulnerabilities

Type of Vulnerability = i	**Severity = Wi**
Off-by-one Error	18
Uncontrolled Memory Allocation	4
Improper Input validation	20
Arithmetic Underflow	4
Improper check for unusual or exceptional conditions	12
Dead Code	3
De-serialization of un-trusted data	7
Incorrect Conversion between Numeric Types	4
finalize() Method Declared Public	4
Improper initialization	1
Absolute Path Traversal	16
Unrestricted File Upload with Dangerous Type	10
Uncontrolled Resource Consumption	14
Manipulating Inputs to File System Calls	3
URL Redirection to Un-trusted Site	3

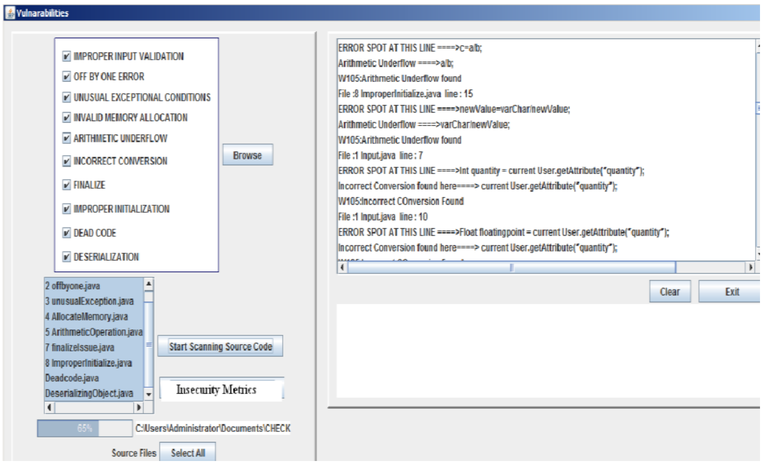
Client Side Enforcement of Server Side Security	4
Sensitive Cookies in HTTPS Session Without 'Secure' Attribute	4
Improper Neutralization of HTTP Headers for Scripting Syntax	1
Information Exposure through Log Files	5

5- WORKING OF SecCheck

SecCheck uses pattern matching methodology to detect weaknesses present in Java applications. The tool takes as input any Java program and scans to identify the vulnerabilities. If any vulnerability is detected then it displays warning message. The steps followed are

- Select the input Java program
- Select from the drop down list types of vulnerabilities to be detected in given input Java program

The tool displays type of vulnerabilities and the place of occurrence as shown in Fig 1. It also gives the Degree of Insecurity in the input program



```
ERROR SPOT AT THIS LINE ====>c=a/b;
Arithmetic Underflow ====>a/b;
W105:Arithmetic Underflow found
File :8 ImproperInitialize.java  line : 15
ERROR SPOT AT THIS LINE ====>newValue=varChar/newValue;
Arithmetic Underflow ====>varChar/newValue;
W105:Arithmetic Underflow found
File :1 Input.java  line : 7
ERROR SPOT AT THIS LINE ====>Int quantity = current User.getAttribute("quantity");
Incorrect Conversion found here====> current User.getAttribute("quantity");
W105:Incorrect COnversion Found
File :1 Input.java  line : 10
ERROR SPOT AT THIS LINE ====>Float floatingpoint = current User.getAttribute("quantity");
Incorrect Conversion found here====> current User.getAttribute("quantity");
```

Figure 1 Front end of SecCheck

It has three functional modules as shown in Figure 2:

Scanner: This module scans each line of source code one by one.

Pattern Matching Module: After scanning, SecCheck compares each line to find out if it contains a set of keywords which makes the program vulnerable to security threats. This is done by matching each line with the list of strings stored in a database.

Display Module: If there is a string match then a warning message is flagged to the user.

After the entire program is scanned, the *Degree of Insecurity* is calculated and displayed.

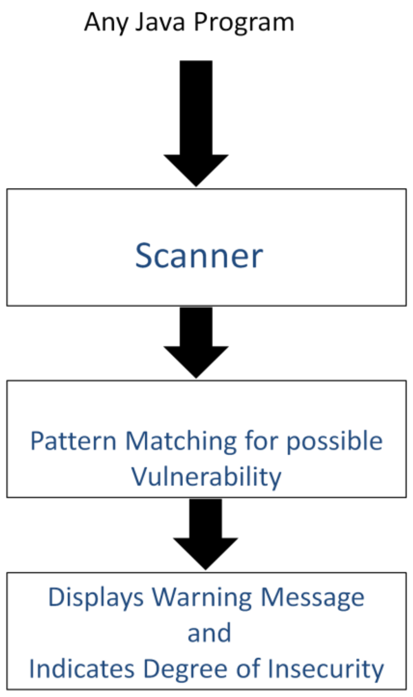

Any Java Program
Scanner
Pattern Matching for possible Vulnerability
Displays Warning Message and Indicates Degree of Insecurity

Figure 2: Architecture of SecCheck

6- EXPERIMENTAL RESULTS

We used the tool SecCheck for detection of vulnerabilities on two classes of Java programs. Table 3 shows the programs written by the professionals taken from different vulnerability tracking sites including few from Common Weakness Enumeration (CWE) [7] site. Table 4 shows the programs written by students of an engineering college.

The results of measurements in these programs are given in Tables 3 and 4 with the vulnerabilities detected and the *Degree of Insecurity* calculated as per the expression in Section 4 of this paper.

Table 3: Degree of Insecurity calculated in Programs written by Java professionals

Name Of Program	**Source**	**Size**	**ISM**
Concurrency	http://stackoverflow.com/questions/6084722/java-threads-concurrency-delta-off-by-one-error	3KB	48
Countdivisor	http://math.hws.edu/javanotes/c3/s4.html	3KB	13
Bouncingline	http://www.java-examples.com/generate-bouncing-lines-using-applet-example	5KB	81
Javapyramid	http://www.java-examples.com/java-pyramid-3-example http://www.java-examples.com/prime-numbers-java-example	2KB	36
Internlengthy	http://howtodoinjava.com/2012/10/31/why-not-to-use-finalize-method-in-java/	2KB	21
Heapmemory	http://javarevisited.blogspot.com/2012/01/find-max-free-total-memory-in-java.html	2KB	75
Splitxmlfile	http://stackoverflow.com/questions/6341203/java-xml-getattribute	3KB	162
Testgc	http://howtodoinjava.com/2012/10/31/why-not-to-use-finalize-method-in-java/	2KB	73
Deserial	http://javabynataraj.blogspot.in/2011/04/what-is-deserialization-in-java-write.html	2KB	52
Inputfilter	http://stackoverflow.com/questions/5003939/dead-code-warning	3KB	100
Average	http://www.cwe.mitre.org	5KB	339
arraydemo	http://www.cwe.mitre.org	7KB	130

DOSattack	http://www.coderanch.com/t/554166/sockets/java/DOS-attack-server	8KB	128
Informationexp	http://www.roseindia.net/java/example/java/util/quintessential-logging-program.shtml	2KB	31
Multipartfile	http://www.tutorialspoint.com/servlets/servlets-file-uploading.htm	3KB	30
Absolutepath	http://www.mkyong.com/java/how-to-read-file-in-java-fileinputstream/	2KB	56
Sessionclient	http://www.coderanch.com/t/565447/Tomcat/HttpSession-client-server	3KB	22
Postredirect	http://www.coderanch.com/t/598162/Servlets/java/configure-sendRedirect	6KB	46
Sensitivecookie	http://java-demos.blogspot.in/2013/04/cookies-in-servlets-with-example.html	5KB	28
Filesystemcall	http://www.java-samples.com/showtutorial.php?tutorialid=8	3KB	85
Average Value of ISM Calculated from these Programs			**77.8**

The programs that contain higher value of *Degree of Insecurity* are said to be more vulnerable and must be corrected. For example programs such as splitxmlfile.java, average.java, arraydemo.java, DOSattack.java exceeds the range of 100, hence they are said to be insecure. Splitxmlfile.java is said to be more vulnerable due to the presence of improper input validation, incorrect conversion between numeric types occurring multiple times. Average.java, arraydemo.java contains dead code, de-serialization of un-trusted data vulnerability multiple times and it contains maximum number of weaknesses to help attackers to misuse them, DOSattack.java is also more vulnerable because it contains Path traversal weakness, multipart file detection, uncontrolled resource consumption problem, URL redirection problem, sensitive cookie and manipulating file system call issues so when there is presence of so many weaknesses in a single program then it gives a clear picture that the program is too weak and it becomes easier for an attacker to attack on resources or with URL redirection so as to redirect user to different unexpected site and the attacker might misuse with user credentials and even uploading any content files and many more weaknesses that can really affect the development of software and make the software insecure.

Table 4: Degree of Insecurity Calculated in Programs written by Students

Program Name	**Size**	**ISM**
ClientA	13 KB	313
Router	30 KB	615
Server	36 KB	756
Dist	31 KB	525
Logger	19 KB	213
Arithmetic	32KB	659
Average Value of ISM Calculated from these Programs		**513.5**

Table 4 describes the experimental results on Java programs written by the students [3][4] along with the *Degree of Insecurity* calculated in each of them.

Comparison of the *Degree of Insecurity* in the two classes reveal that the Java programs written by students have a higher *Degree of Insecurity* than those written by professional programmers and therefore are more prone to security attacks. This was in line with our expectations.

7- CONCLUSION

Vulnerabilities in software are weaknesses caused by defects in design and code and have to be removed to make it more secure. Detecting such vulnerabilities manually is painstaking and it is necessary to have tools that can help programmers to detect and correct these during development stage.

There are a number of tools available to detect the vulnerabilities present in application programs written in various programming languages. But these tools detect only few vulnerabilities which are very common and do not calculate the Degree of Insecurity.

The tool developed by the authors and described in this paper detects nineteen vulnerabilities in Java source code and also calculates the *Degree of Insecurity* in the application. The effectiveness of the tool has been demonstrated by applying it on two classes of programs: one by professionals and the other by students and results were as per expectations.

Table 1: Comparison with Available Vulnerability Detection Tools

Tool	Developed by	Features	Languages	Remarks
bugScout[27]	buguroo	Multiple security failures, such as deprecated libraries errors, vulnerable functions, sensitive information within the source code comments, etc.	Java, C#, Visual Basic, ASP, php	It is a commercial tool provided on SaaS in the cloud. It does not calculate security metrics.
Jtest[28]	Parasoft	Defects such as memory leaks, buffer issues, security issues and arithmetic issues, plus SQL injection, cross-site scripting, exposure of sensitive data and other potential issues	Java	It is a commercial tool Does not find all vulnerabilities. Does not calculate security metrics.
Checkmarx[29]	Checkmarx	Cover all known OWASP and SANS vulnerabilities and comply with PCI and other standards. Includes a query language that enables infinite customization and detection accuracy with virtually zero false positives.	Java, C#/.NET, PHP, C, C++, Visual Basic 6.0, VB.NET, Flash, APEX, Ruby, JavaScript, ASP, Android, Objective C, Perl	It is a commercial tool. Converts all languages code and flow into a single, common-language format stored in a persistent database. Does. not calculate security metrics
CodeSecure[30]	Armorize Technologies	XSS, SQL Injection, Command Injection, tainted	ASP.NET, C#, PHP, Java, JSP, VB.NET, others	It is a commercial tool. Does not calculate security metrics. Developed

		data flow, etc.		only for web application security. Performs data flow and control flow analysis on each line of code.
Coverity[31] SAVE™	Coverity	Flaws and security vulnerabilities - reduces false positives while minimizing the likelihood of false negatives.	C, C++, Java, C#	It is a commercial tool. Does not find all vulnerabilities. Does not calculate security metrics.
FindBugs[32] FindSecurityBugs	Bill Pugh and David Hovemeyer	Null pointer deferences, synchronization errors, vulnerabilities to malicious code, etc. It can be used to analyse any *JVM languages*, more security detectors (Command Injection, XPath Injection, SQL/HQL Injection, Cryptography weakness and more).	Java, Groovy, Scala	Operates on Java byte code, rather than source code. Does not calculate security metrics
Fluid[33]	Lockheed Martin's Software Technology Initiative (STI)	"Analysis based verification" for attributes such as race conditions, thread policy, and object access with no false negatives	Java	Checks for concurrency errors. Does not calculate security metrics
HP QAInspect[34]		Application vulnerabilities	C#, Visual Basic,	Mimics real hacking techniques

	HP		JavaScript, VB Script	and attacks, enabling to analyze web applications and services for security vulnerabilities. Does not calculate security metrics
Insight[35]	Klocwork	Buffer overflow, un-validated user input, SQL injection, path injection, file injection, cross-site scripting, information leakage, weak encryption and vulnerable coding practices, as well as quality, reliability and maintainability issues.	C, C++, Java, and C#	Analyses web applications. Does not calculate security metrics
Jlint[36]	Konstantin Knizhnik	Bugs, inconsistencies, and synchronization problems	Java	Performs data flow analysis on the code and builds the lock graph to check vulnerabilities. Does not calculate security metrics.
LAPSE[37]	OWASP	Helps audit Java J2EE applications for common types of security vulnerabilities found in Web applications.	Java	Difficulty arises when applications consisting of thousands of lines of code or having a complex structure with many Java

				classes. Does not calculate security metrics
PMD[38]		Questionable constructs, dead code, duplicate code	Java	Does not calculate security metrics. Does not find all vulnerabilities.
QA-J[39]	PRQA programming Research	A suite of static analysis tools, with over 1400 messages. Detects a variety of problems from undefined language features to redundant or unreachable code.	Java	Deep, accurate, high-fidelity parsing, and incorporates a sophisticated solver-based dataflow engine to identify intricate value-tracking issues and coding vulnerabilities. Does not calculate security metrics.
Resource Standard Metrics (RSM)[40]	M Squared Technologies	Scan for 50 readability or portability problems or questionable constructs, e.g. different number of "new" and "delete" key words or an assignment operator (=) in a conditional (if).	C, C++, C#, and Java	Measures code quality and metrics. Does not calculate security metrics.
Rational AppScan Source Edition[41]	IBM (formerly Ounce Labs)	Coding errors, security vulnerabilities, design flaws,	C, C++, Java, JSP, ASP.NET, VB.NET, C#	Does not calculate security metrics. Tests

		policy violations and offers remediation		web applications. Does not find all vulnerabilities.
SCA[42]	Fortify Software	Security vulnerabilities, tainted data flow, etc. "more than 470 types of software security vulnerabilities"	ASP.NET, C, C++, C# and other .NET languages, COBOL, Java, JavaScript/AJAX, JSP, PHP, PL/SQL, Python, T-SQL, XMLand others	Does not calculate security metrics. Does not find r all vulnerabilities
Microsoft Office Security Assessment Tool [43]	Microsoft	Assess weaknesses in their current IT security environment, create a prioritized list of issues, and help provide specific guidance to minimize those risks..		Does not calculate security metrics.
Nessus[44]	Tenable Network Security	Remote and local (authenticated) security checks,	Unix	Does not check security metrics. Does not provide the scanning in the programs running in the machines. Checks hardware weaknesses.
Core Impact[45]	IBM	SS Internet Security vulnerability scanner		Does not check security metrics isn't cheap. It sports a large, regularly updated database of professional

				exploits,
X-scan[46]	Security Auditor's Research Assistant[SARA] tool	Detecting service types, remote OS type/version detection, weak user/password pairs.		Does not calculate security metrics. Checks for network weakness only

ACKNOWLEDGEMENTS

The work reported in this paper has been done with the support of the Department of Science and Technology, Government of Karnataka, India under VGST Scheme.

The authors wish to acknowledge the assistance of S P Srinivas, Sowmya K B and Nirja Parida of East Point College of Engineering and Technology, Bangalore in preparing the manuscript.

REFERENCES

[1] G. Mcgraw, Software Security: Building Security In, Addison Wesley, 2006.

[2] A. K.Talukder, M. Chaitanya. Architecting Secure Software Systems, Auerbach Publications, 2009.

[3] R. Priyadarshini, A. Basu and S. Sushma, "SecCheck: A Tool to Detect Vulnerabilities in Java Code," International Conference on On-Demand Computing, ICDOC Bangalore, Nov 15-16, 2012.

[4] N. Ghosh and A. Basu, "WebCheck: A Tool to Detect Weaknesses in Java Web Applications," International Conference on Information and Communication Engineering ICICE Bangalore, June 28-29, 2013.

[5] A. Mammar, A. Cavalli, W. Jimenez, W. Mallauli, and E. M. deOca, "Using testing techniques for vulnerability detection in C programs," Proceedings of the 23rd IFIP WG 6.1 International Conference on Testing Software and Systems, ICTSS'11, pp. 80-96, 2011.

[6] V. B. Livshits and M. S. Lam, “Finding Security Vulnerabilities in Java Applications with Static Analysis,“ Proceedings of the 14th Conference on USENIX Security Symposium, CA, pp.271-286, 2005.

[7] http://cwe.mitre.org

[8] Off by one error: http://cwe.mitre.org/data/definitions/193.html

[9] Uncontrolled Memory Allocation:

http://cwe.mitre.org/data/definitions/789.html

[10] Improper Input Validation: http://cwe.mitre.org/data/definitions/20.html

[11] Improper Check for unusual or exceptional conditions:

http://cwe.mitre.org/data/definitions/754.html

[12] Arithmetic Underflow:http://en.wikipedia.org/wiki/Arithmetic_underflow

http://javapapers.com/core-java/java-overflow-and- underflow/

[13] Dead Code: http://en.wikipedia.org/wiki/Dead_code

[14] Deserialization of Untrusted Data

http://cwe.mitre.org/data/definitions/502.html

[15] Incorrect Conversion between Numeric Types

http://cwe.mitre.org/data/definitions/681.html

[16] finalize() Method Declared Public:

http://cwe.mitre.org/data/definitions/583.html

[17] Improper Initialization: http://cwe.mitre.org/data/definitions/665

[18] Absolute Path Traversal:http://cwe.mitre.org/data/definitions/36.html

[19] Uncontrolled Resource Consumption(“Resource Exhaustion”):

http://cwe.mitre.org/data/definitions/400.html

[20] Unrestricted Upload of File with Dangerous

Type:http://cwe.mitre.org/data/definitions/434.html

[21] Manipulating Inputs to File System Calls:

http://capec.mitre.org/data/definitions/76.html

[22] URL Redirection to Untrusted Site(‘Open Redirect’):

http://cwe.mitre.org/data/definitions/601.html

[23] ClientSide Enforcement of ServerSide Security:

http://cwe.mitre.org/data/definitions/602.html

[24] Sensitive Cookie in HTTPS Session Without 'Secure' Attribute:

http://cwe.mitre.org/data/definitions/614.html
[25] Improper Neutralization of HTTP Headers for Scripting Syntax:http://cwe.mitre.org/data/definitions/644.html
[26] Information Exposure through Log Files:http://cwe.mitre.org/data/definitions/532.html
[27] https://www.buguroo.com/en/products/bugscout/
[28] http://www.parasoft.com/jsp/products/jtest.jsp
[29] http://www.checkmarx.com/
[30] http://www.armorize.com/codesecure/
[31] http://www.coverity.com/
[32] http://findbugs.sourceforge.net/
[33] http://fluid-software.com/
[34]https://download.spidynamics.com/products/qainspect/hp/qainspectqcreleasenotes.txt
[35] http://www.klocwork.com/products/insight/
[36] http://jlint.sourceforge.net/
[37] https://www.owasp.org/index.php/OWASP_LAPSE_Project
[38] http://pmd.sourceforge.net/
[39] http://www.programmingresearch.com/
[40] http://msquaredtechnologies.com/
[41] http://www-03.ibm.com/software/products/en/appscan-source/
[42]http://www8.hp.com/us/en/software-solutions/application-security/index.html
[43] http://www.microsoft.com/en-in/download/details.aspx?id=12273
[44] http://www.tenable.com/products/nessus
[45] http://www.coresecurity.com/core-impact-pro
[46] http://www.x-scan.eu/