Deploying Security-Aware Software Systems Using Source Code Vulnerabilities Analysis

J.N. Obidinnu & O.S. Duke
Department of Computer Science
Cross River University of Technology
Calabar, Nigeria
obijulius@yahoo.com

ABSTRACT
This paper identifies the existence of vulnerability in the source code of deployed software as the root cause of majority of the software security problems, and therefore, discusses the vulnerability concept and its severe negative impacts to programmers and users alike. Arising from the discussions, the paper posits that, since the deployed software products are generated from the source code, then the security considerations in all the other stages of the Software Development Life Cycle (SDLC) can only be meaningful if their functionalities are evident in the source code. And further arising from this position is that the programmers are in a better position to detect and remove vulnerabilities more than the other operators in the SDLC, but majority are either highly unknowledgeable or ill-equipped to perform this function. And in order to close this gap, the paper identifies the attackers' tools and their characteristics, and recommends that the programmers (defenders) proactively use them on their source code before deployment. It is our position that adopting this recommendation will empower the programmer towards the deployment of security aware software systems, which will in turn; make attackers spend much more time and resources trying to conduct their malicious acts. And spending more time and resources could evoke the fear of being noticed, and be discouraged from attacking such system. The attacker might just decide to look for easier targets for the feeling of success.

Keywords: Vulnerability, Software Exploit, Source Code Security Analysis, Software Security, Security-Aware

1. INTRODUCTION
Software exploit is the process of gaining illegal access to restricted computer information, by supplying inputs that will make the software system behave in an undesired manner. Consequently, when a software system is exploited, the Confidentiality, Integrity and Availability (CIA) of the system and its associated resources could be compromised [15].

Rice in [21] indicates that the reported number of software exploits leading to security breaches is increasing on a yearly basis. However, these exploits are on the rise, in spite of the huge expenditures being made on "security solutions" such as; firewalls, and antivirus systems [8]. Antivirus systems, for instance, concentrate on the detection and remediation of known virus patterns (signatures) that have been attacked previously, leaving unknown patterns in the programs for further exploration; firewalls, on the other hand, provide perimeter defense mechanisms, but they must also allow traffic into the application, and this could enable an attacker to take advantage of security weaknesses in code.

In the same way, [12] asserts that, software exploits are on the rise because attackers are becoming increasingly aware of the existence of security vulnerabilities in deployed software. And [18] posits that, if there was exploit, then there is a strong possibility that vulnerability exists in the source code of the exploited software. Hoglund and McGraw in [8] affirm this position when it refers to vulnerability, as the root cause of most software security problems. And [24] reports that, in about sixty-four per cent of the times, vulnerabilities exist due to programming errors.

Losses due to these exploits have severe negative impacts to developers and users alike, such as; denial of service, theft of credit card details and other private information, theft of bank’s customers information, reduction in customers’ trust to use such lines of products, drop in reputation of the software company, losses of jobs, losses in wages, and many others [26], [11], and [21].
In view of the above, this paper discusses the vulnerability concept and its implications, so as to raise the consciousness of software developers in proactively addressing source code vulnerabilities in their software systems before deployment. It is our position that this will have positive impact in crafting security-aware software systems, which will in turn, make the attacker spend much more time and resources trying to conduct the malicious act. And spending more time and resources could evoke the fear of being noticed, and be discouraged from attacking such system. The attacker might just decide to look for easier targets for the feeling of success.

2. THE VULNERABILITY CONCEPT

The security weaknesses in source code of a software system constitute its vulnerability set, which attackers exploit in gaining access to restricted information [25]. The vulnerability concept is better illustrated using the cause and effect diagram presented in Figure 1.

![Figure 1: Illustration of the Vulnerability](Image)

In [14], an Error, bug or flaw is defined as a mistake made by a developer, by reasoning or doing things incorrectly when trying to solve a problem. More so, an error becomes a Fault when it is written (included) in any part of a deployed software product. A Fault becomes Vulnerability when it is discovered as an opening by an attacker. An Exploit or attack is an action that takes advantage of the presence of Vulnerability to affect a system negatively. Failure is a state of a software system when it behaves in an inconsistent manner at the injection of certain input strings. At this state, the software system generates outputs different from the desired outputs originally intended by the developers. When this happens, attackers could completely take over an application, steal data, or prevent the software from working at all.

Vulnerabilities exist in all types of software, written in all types of programming languages [35]. They exist in software systems, partly because, majority of the programmers, often tend to address security in terms of attaching “security features”, such as cryptographic ciphers, passwords, and access control mechanisms, with the belief that, if these tools function correctly, then system security will be ensured [2]. However, [3] maintains that: “Software Security” is not “Security Software”. Therefore, using one or more security features in a product does not ensure security, and might portend danger, because security is a process, and not a product.

Security issues can be considered at the different levels of the Software Development Life Cycle (SDLC). However, since the final software product is generated from the source code, Binkley in [1] insists that all security considerations at all levels of the SDLC can only be meaningful if their functionalities are evident in the source code. In confirmation of this position [31] asserts that, if requirements, design, and other earlier software development stages are correct, but are wrongly coded, then all those correctness would have been lost through the source code.

The foregoing confirms the importance of addressing vulnerability issues at the source code level. The analysis of an application’s source code is very essential in the study of its behaviour, in order to obtain a comprehensive understanding of its implementation. And the programmer is in a better position to perform these tasks. However, majority of the programmers are not capable of identifying vulnerable code patterns in their software [33]. Consequently, they relinquish this responsibility to the compilers. However, [10] posits that compilers cannot be relied upon to perform very sophisticated detection of security faults, as security vulnerable code patterns are part of the faults that many compilers are yet capable of detecting [35]. The end-result is that the vulnerable code patterns are unknowingly allowed to be part of the finished product.

Another dimension to this problem is that many programmers are in a hurry to deploy software. They rather prefer to “sell today and fix later” or “I’d rather have it wrong than have it late”. Consequently, patches are produced whenever attacks on their software are publicised. However, much loss would have been incurred before the patches come in. Even so, [12] confirms that majority of the reported vulnerabilities did not have patches after a long while.

3. CONSEQUENCES OF DEPLOYING VULNERABILITY-PRONE SOFTWARE

An attack on a Web site that closes down an e-commerce site can have disastrous consequences for a Web-based business. An intrusion that results in the theft of millions of credit card numbers from an online vendor can result in significant financial loss and, more broadly, reduce consumers’ willingness to engage in e-commerce. An instance of SQL injection vulnerability was exploited in a bank’s web interface in 2007, resulting in the exposure of 3000 customer records out of which 20 was actually stolen before it was discovered and shut down [28]. With such information, attackers could use the stolen names, addresses, dates of birth and bank account numbers of the people to defraud the banks of huge sums of monies [20].
The Code Red virus incident in 2001 as well as the Lovebug virus in 2000 occurred due to the exploitation of vulnerabilities in software. Cisco Systems Inc. in [3] reports that the damage due to Code Red and Lovebug were estimated at $2.62 billion and $8.75 billion respectively. Telang and Wattal in [26] also reports of a National Institute of Standards and Technology study, whose outcome, estimates the cost of faulty software at $60 billion per year. Patton in [19] reports that, Intel paid more than $400 million with apologies to users, for the replacement of chips, whose errors were identified before deployment, but was ignored because of the cost of fixing the errors at that stage. Intel knows better now. They have acquired McAfee, a leading security software company to support security issues [29]. Similarly other major hardware manufacturing companies are doing same; IBM acquired Ounce Labs in 2009 [13], while HP also acquired Fortify (another leading security software company) in 2010 [6].

Instances exist where software developers also seem to suffer due to security errors in their products. The report in [26] further indicates that, in addition to the voluminous amount of bad press that security vulnerabilities generate for software companies, a public software company’s valuation drops after the announcement of any vulnerability. A National Institute of Standards and Technology report, cited by [11], confirms that, fixing application error(s) after deployment, costs approximately 30 times more than addressing it during the design phase. This is shown in Table 1.

Table 1: Relative cost of fixing bugs at different stages of software development (Source: [11])

<table>
<thead>
<tr>
<th>Type of error</th>
<th>Design</th>
<th>Coding</th>
<th>Integration</th>
<th>Beta</th>
<th>Deployment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design</td>
<td>1x</td>
<td>5x</td>
<td>10x</td>
<td>15x</td>
<td>30x</td>
</tr>
<tr>
<td>Coding</td>
<td>1x</td>
<td>10x</td>
<td>20x</td>
<td>30x</td>
<td>30x</td>
</tr>
<tr>
<td>Integration</td>
<td>1x</td>
<td>10x</td>
<td>20x</td>
<td>30x</td>
<td>30x</td>
</tr>
</tbody>
</table>

Table 1 indicates that, fixing a design error after software deployment, costs approximately 30 times more than addressing it during design. And these estimates do not factor in costs such as losses in market share reputation or customer satisfaction.

Figure 2 is a graphical representation of the costs associated with fixing a bug earlier or later during the software development stages, from where the size of a bug is an indication of how expensive it will be to fix it at that stage.

The idea behind Table 1 and Figure 2 is that, the earlier, in the software development lifecycle that security vulnerabilities are detected and rectified, the cheaper and less risky the software becomes. Consequently, in order to help prevent expensive fixes, enterprises are encouraged to build application security testing approaches into their development and delivery processes.

Developers are also blamed for software security mishaps and punished through losses in wages or loss of jobs from their employers. So far, only the attacker caught on the act, may be charged in court. In the future, software companies may be charged for not preventing the attacks. In the middle of January 2002 the discussion about responsibility for security intrusions took an interesting turn. For instance, [36] reports that, the United States National Academies released a prepublication, recommending policy-makers to create laws that would hold companies accountable for security breaches resulting from vulnerable products. This stresses the importance of helping software developers produce more secure software.
This paper adopts the above recommendation, and introduces an approach which recognises that an attacker needs to find only one vulnerable code pattern to perform an exploit, while the defenders (developers) need to find and fix all, or otherwise defend against the eventual attacks. And since the security of a software system cannot be localised to a specific place in the software, the developer obviously has more work to do. Fortunately, majority of the attackers’ tools are freely available. Unfortunately, the developers’ have often ignored the proactive use of these tools, but rather choose to be reactive after exploits. This paper therefore, chronicles the attackers’ tools and their characteristics, aimed at empowering the software developers towards detecting security vulnerabilities in their source code before deployment.

4. RECOMMENDED TOOLS FOR SOURCE CODE VULNERABILITY CHECKS

Hoglund and McGraw in [8] posit that, the path to creating a secure application begins by rigorously testing source code for all vulnerabilities and ensuring that the use of the application does not allow for the compromise of data privacy and integrity. A good starting point would be tools that can be applied directly to the source code to solve or warn about security vulnerabilities. This implies detecting and resolving the problems by the developer before someone else (the malicious user) does. This informs the approach in this paper, where we present a description of the common tools that could be used for detecting security vulnerable code patterns in source code.

The tools are called source code security analysers. For the purpose of this paper, we select five of them. Such tools are programming language dependent, and our selected tools have capacity to analyse C/C++ programs. We select C++ because it contains so many unsafe functions [22]. Consequently, they contribute very significantly to the very high number of reported security vulnerabilities in deployed systems. To confirm this, we did a study of the listing in [16], which is one of the most authoritative sources of vulnerability reports.

Our study compares the contributions of four referenced programming languages; C++, Java, PHP, and Perl, to the reported vulnerability population. Using the “Weaknesses by language” criterion, the languages scored the following “High’s”: C++=17; Java=9; PHP=14; and Perl=11, where a rating of “High” indicates that the referred weakness frequently appears in software written in the language, and it is often a security problem instead of “just a bug”. This scoring clearly puts C++ ahead of others, in terms of contributing highly to the reported number of vulnerability population ravaging deployed systems. In the following sections, we describe the operations of the selected source code security analysis tools.

4.1 Characteristics of Source Code Security Analysis Tools

Source code analysis, also called, static analysis, is the process of extracting information about a program from its source code. The extracted information must be consistent with language semantics and should help a programmer gain insight as to the source code’s meaning. They gather the information from a program without executing it. Source code security analysers provide very valuable results by pinpointing each security vulnerability at the precise line of source code and detailing information about the type of fault, and how to fix it.

NIST in [17] recommends that, at a minimum, a source code security analysis tool should:

- Identify a set of classes of software security weaknesses in source code,
- Report the security weaknesses that it identifies, what kind of weakness each one is, and where each one is located, and
- Not have many false positives.

The selected source code security analysis tools include: IT'S4, FlawFinder, RATS, SPLINT, and BOON.

Table 2 is a representation of the types of faults that the selected tools can detect. The information in the table substantially derives from [18]. From the table, Tool Name is the name of the tool. Language represents the type of language (s) the tool name on the same row could be used to check. Availability implies the commercial status of the tool. Type of fault (s) checked for, represents the kinds of faults that the tool is capable of checking. Finally, the last column is the date such tool was made publicly known.
### Table 2: Representative types of Static Analysis Tools and features

<table>
<thead>
<tr>
<th>Tool Name</th>
<th>Language(s)</th>
<th>Availability</th>
<th>Type of fault(s) checked for:</th>
<th>Date of Release</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flawfinder</td>
<td>C/C++</td>
<td>Free</td>
<td>Uses of risky functions, buffer overflow (strcpy()), format string ([v][f]printf()), race conditions (access(), chown(), and mktemp()), shell metacharacters (exec()), and poor random numbers (random()).</td>
<td>2006</td>
</tr>
<tr>
<td>RATS (Rough Auditing Tool for Security)</td>
<td>C</td>
<td>Free</td>
<td>Potential security risks</td>
<td>2005</td>
</tr>
<tr>
<td>ITS4</td>
<td>C, C++</td>
<td>Free for non-competing uses</td>
<td>Potentially dangerous function calls, with risk analysis of some</td>
<td>2008</td>
</tr>
<tr>
<td>Splint</td>
<td>C</td>
<td>Free</td>
<td>Security vulnerabilities and coding mistakes. With annotations, it performs stronger checks.</td>
<td>2007</td>
</tr>
<tr>
<td>BOON</td>
<td>C</td>
<td>Free</td>
<td>Integer range analysis, determines if an array can be indexed outside its bounds.</td>
<td>2009</td>
</tr>
</tbody>
</table>

Being publicly available makes the tools more accessible for public assessment. For each of them, we present the following features: where to obtain it, when it was released, the approach used for the analysis, the output presentation, and benchmark characteristics where available. In the following sections, we present these features as they relate to the listed tools.

### 4.2 ITS4

It's the Software Stupid Security Scanner (ITS4) was released in 2000 by Cigital for detecting security vulnerabilities [30]. ITS4 is a lexical analyzer for statically scanning C and C++ source code for function calls that are known to cause vulnerabilities. It searches for functions that are commonly involved in implementation flaws.

The goal of ITS4 is to focus on the person performing a vulnerability analysis. Instead of having an analyst search through an entire program, ITS4 provides an analyst with a list of potential trouble spots. Something similar can be done with grep. However with grep, you need to remember what to look for every single time. ITS4 already contains knowledge about what to look for.

ITS4 also performs some basic analysis to try to rule out conditions that are obviously not problems. For example, `printf()` is a frequently misused function, but if the format string is constant it is no problem and should not be reported. ITS4 prioritizes the list of vulnerabilities it finds which is of course helpful; the most severe vulnerability is at the top of the list. ITS4 looks for several builds of the software. In C language there are usually pre-processing directives that leave out certain parts of the software during compilation. The cited authors above affirm that ITS4 circumvents this condition to analyze every piece of code in the software.

This is done for several reasons. It avoids the complexities of real parsing, so it can be used in an integrated development environment to highlight potential errors from within an editor.

**Input:**
Plain C or C++ source code

**Output:**
```
$ its4 fingerd.c
fingerd.c:112:(Urgent) fprintf
Non-constant format strings can often be attacked.
Use a constant format string.
--------------
fingerd.c:91:(Risky) execv
Many potential problems. Close all fds, clean the environment, set the umask to something good, and reset uids before calling.
--------------
fingerd.c:97:(Risky) fdopen
Can be involved in a race condition if you open things after a poor check. For example, don't check to see if something is not a symbolic link before opening it. Open it, then check by querying the resulting object. Don't run tests on symbolic file names...
Perform all checks AFTER the open, and based on the returned object, not a symbolic name.
--------------
fingerd.c:99:(Some risk) getc
Be careful not to introduce a buffer overflow when using in a loop.
Make sure to check your buffer boundaries.
```
This output from the command-line interface prints the found problems with filename, line-number and found function-name. The results are first sorted for severity, then for function-names. A short description of the problem and possible solutions are displayed. The sort order can be changed with command-line parameters to ITS4. The vulnerability database contains the most often misused C functions.

### 4.3 FlawFinder

Flawfinder from [34] is a lexical analyzer that operates on source code. During analysis, FlawFinder first tokenizes the source code and then matches the token stream against a library of vulnerable functions.

FlawFinder searches through C/C++ code looking for potential security flaws. After the code analysis is complete, it produces a list of potential flaws sorted by risk. The database of FlawFinder contains both general rules that effect any program and specific Windows and Unix functions that are very vulnerable to exploitation [27]. FlawFinder also shows some intelligence when it comes to scanning for vulnerabilities.

For example, in tests using intentionally insecure code, FlawFinder was able to distinguish between strcpy() from a constant sized string and variable length strings and tell the difference between vulnerabilities and false hits [36]. One of the major problems with FlawFinder is the lack of any pre-processing, so no macros or definitions are expanded, and no external functions available in source form are examined. This design decision limits the abilities of FlawFinder.

**Input:**
Plain C/C++ source code

**Output:**
Flawfinder prints potential security flaws, sorted by risk. It prints the filename and line-number of the potential problems, the risk value, and the category of the problem and the name of the identified function. It then explains why this function often is a problem.

Flawfinder also displays a summary over the complete run, printing the total number of hits, lines analyzed, SLOC (Source Lines Of Code) found, hits sorted by risk, and hits per SLOC.

**Example:**
Number of dangerous functions in C/C++ ruleset: 158
Examing example7.c
example7.c:36: [4] (format) printf:
If format strings can be influenced by an attacker, they can be exploited. Use a constant for the format specification.
example7.c:17: [2] (buffer) char:
Statically-sized arrays can be overflowed. Perform bounds checking, use functions that limit length, or ensure that the size is larger than the maximum possible length.
example7.c:24: [1] (buffer) strncpy:
Easily used incorrectly; doesn't always \0-terminate or check for invalid pointers. Risk is low because the source is a constant string.
example7.c:26: [1] (buffer) strncpy:
Easily used incorrectly; doesn't always \0-terminate or check for invalid pointers.

Hits = 4
Lines analyzed = 43 in 0.54 seconds (1210 lines/second)
Physical Source Lines of Code (SLOC) = 25
Minimum risk level = 1

### 4.4 RATS

Rough Auditing Tool for Security (RATS) is a lexical analysis tool from [23]. As opposed to ITS4, RATS not only scans C and C++ code but also Perl, PHP and Python source code and flags common security bugs such as buffer overflows and race conditions. Like FlawFinder and ITS4, RATS has a database of vulnerabilities and sorts found security bugs by risk. RATS is invoked from a shell with source code as input. It traverses the code and produces output with risk grading and short descriptions of the potential problems. This is very similar to the approaches taken by ITS4 and FlawFinder.

**Input:**
Plain C/C++/Perl/PHP/Python source code. The files need to be named appropriate (.c, .cc, .pl, .php, .py) to allow the usage of the correct rule set (Example, the Perl rule set). Alternatively the tool’s language option must be set, to enforce the usage of a specified rule set.
Output:
RATS prints potential problems sorted by severity, then by function name, then by file, then by line number. For each function name rats prints an explanation of the problem (if available in the vulnerability database). Finally Rats prints the number of lines analyzed and the time used.

Example:
Analyzing example7.c
example7.c:17: High: fixed size local buffer
Extra care should be taken to ensure that character arrays that are allocated on the stack are used safely. They are prime targets for buffer overflow attacks.

element7.c:36: High: printf
Check to be sure that the non-constant format string passed as argument 1 to this function call does not come from an untrusted source that could have added formatting characters that the code is not prepared to handle.

Total lines analyzed: 44
Total time 0.000395 seconds
111392 lines per second

4.5 SPLINT

Lint is one of the first and most widely used static analysis tools for C and C++. Lint, initially released for UNIX environments by Bell Labs, checks programs for a large set of syntax and semantic errors. However, [7] released a version that operates in DOS, Windows, and OS/2 environments, as well as with UNIX. Newer versions of Lint include value tracking, which can detect subtle initialization problems, and inter-function value tracking, which tracks values across function calls during analysis and strong type checking. Lint can also validate source code against common safer programming subsets, such as the MISRA C standards and the Scott Meyers’ Effective C++ series of standards.

Lint also supports code-portability checks, which can verify that there are no known portability issues with a given set of source code. Several add-on companion programs can aid in the execution of the Lint program. One such tool, ALOA, automatically collects a set of metrics from the Lint execution that can help in doing a source-code QA check. ALOA provides an overall lint score, as well as break downs by source-code module of the number and severity of faults discovered. ALOA is available under the GPL and can be embedded into several integrated development environments.

Secure Programming Lint (SPLINT), from [5] is a relatively new member of the lint-family. SPLINT is a tool for checking C programs for security vulnerabilities and programming mistakes. It is an enhancement of the LCLint, as described in [9], and has been designed in such a way that it searches specifically for security bugs.

The Splint approach is based on programmer supplied comments in the source code. While this is recommended, it is not required. Splint already contains its own annotated library, which we use for analysis. Because Splint analyzes the parse tree rather than the token stream of the source code, it is potentially better at differentiating between correct and incorrect use of functions than lexical analyzers such as ITS4, FlawFinder and RATS [36].

Input:
Splint needs plain C source code including needed headers.

Output:
Splint parses the given sourcefiles and performs the desired checks or tries to solve constraints on buffer bounds when desired. It then displays the results:

~$ splint +bounds buf.c
Splint 3.1.1
buf.c: (in function main)
buf.c:5:2: Likely out-of-bounds store:
strcpy(dst, src)
Unable to resolve constraint:
requires 4 >= 9
needed to satisfy precondition:
requires maxSet(dst @ buf.c:5:9) >=
maxRead(src @ buf.c:5:14)
derived from strcpy precondition:
requires maxSet(<parameter 1>) >=
maxRead(<parameter 2>)
A memory write may write to an address beyond the allocated buffer.
(Use -likely-boundswrite to inhibit warning)

Finished checking --- 1 code warning

Here Splint has found a likely buffer overflow in function main in file buf.c at line 5 column 2. Splint was unable to resolve the constraint to satisfy the precondition for the C library function strcpy, i.e. the destination buffer is at least as big as the source buffer. The actual used buffers are noted in the precondition as “dst” at line 5 column 9 and “src” at column 14.
4.6 BOON

BOON from [32], implies Buffer Overrun detectiON. Under the assumption that most buffer overflows are in string buffers they model string variables (that is, the string buffers) as two Properties: the allocated size, and the number of bytes currently in use. Then all string functions are modelled in terms of their effects on these two properties of the string variable. The constraints are solved and matched to detect inconsistencies similarly to Splint.

Before analyzing the source code you have to use the C preprocessor on it to expand all macros and #include's. Then BOON parses the code and reports any detected vulnerabilities as belonging to one of three categories, namely \Almost certainly a buffer overflow\", \Possibly a buffer overflow\" and \Slight chance of a buffer overflow\". The user needs to go check the source code by hand and see whether it is a real buffer overflow or not. Note that BOON does not detect format string vulnerabilities and is thus not tested for that.

BOON applies integer range analysis to determine whether a C program can index an array outside its bounds [32]. While capable of finding many errors that lexical analysis tools would miss, the checker is still imprecise: It ignores statement order, it can't model inter-procedural dependencies, and it ignores pointer aliasing.

Input:
BOON needs preprocessed C source code. It provides a script "preproc":

```
boon-1.0$ preproc examples/main.c
```

This call produces the file examples/main.i.

Usage:
Call "boon" from its source-directory with the preprocessed .i files:

```
boon-1.0$ boon examples/main.i
```

BOON then parses the file and generates and solves the constraints.

Output:
BOON displays the runtime for parsing the source and solving the constraints.
It then lists possible vulnerabilities:

```
POSSIBLE VULNERABILITIES:
Possibly a buffer overflow in `x@main()':
10..10 bytes allocated, 0..19 bytes used.
<= siz(x@main())
<= len(x@main())
```

Here BOON has found a possible buffer overflow in the buffer "x". It has determined that there may be up to 19 bytes written to a buffer of size 10.

5. CONCLUSIONS

This paper exposes the unrelenting characteristics of hackers in ensuring that the security protections provided for software systems are compromised. And when these protections become compromised, the Confidentiality, Integrity and Availability (CIA) of the system and its associated resources fall prey to the whims and caprices of the hackers.

This paper identifies the existence of vulnerability in the source code of deployed software as the root cause of majority of the software security problems, and therefore, discusses the vulnerability concept and its severe negative impacts to programmers and users alike. Arising from the discussions, the paper posits that, since the deployed software products are generated from the source code, then the security considerations in all the other stages of the Software Development Life Cycle (SDLC) can only be meaningful if their functionalities are evident in the source code. And further arising from this position is that the programmers are in a better position to detect and remove vulnerabilities more than the other operators in the SDLC, but majority are either highly unknowledgeable or ill-equipped to perform this function.

And in order to close the existing gap, the paper identifies the attackers' tools and their characteristics, and recommends that the programmers (defenders) proactively use them on their source code before deployment. It is our position that adopting this recommendation will empower the programmer towards the deployment of security aware software systems, which will in turn; make attackers spend much more time and resources trying to conduct their malicious acts. And spending more time and resources could evoke the fear of being noticed, and be discouraged from attacking such system. The attacker might just decide to look for easier targets for the feeling of success.
REFERENCES


