An automated approach to fix buffer overflows

Aamir Shahab[1], Mamdouh Alenezi[2], Muhammad Nadeem[3], Raja Asif[4]
[1,3,4]Department of Computer Engineering, Faculty of Information and Communication Technology,
Balochistan University of Information Technology, Engineering and Management Sciences (BUITEMS), Pakistan
[2]College of Computer and Information Sciences, Prince Sultan University, Saudi Arabia

Article Info

Article history:

Received Oct 23, 2019
Revised Jan 16, 2020
Accepted Feb 2, 2020

Keywords:

Buffer overflows
Public vulnerability repository
Software vulnerabilities
Static analysis

ABSTRACT

Buffer overflows are one of the most common software vulnerabilities that occur when more data is inserted into a buffer than it can hold. Various manual and automated techniques for detecting and fixing specific types of buffer overflow vulnerability have been proposed, but the solution to fix Unicode buffer overflow has not been proposed yet. Public security vulnerability repository e.g., Common Weakness Enumeration (CWE) holds useful articles about software security vulnerabilities. Mitigation strategies listed in CWE may be useful for fixing the specified software security vulnerabilities. This research contributes by developing a prototype that automatically fixes different types of buffer overflows by using the strategies suggested in CWE articles and existing research. A static analysis tool has been used to evaluate the performance of the developed prototype tools. The results suggest that the proposed approach can automatically fix buffer overflows without inducing errors.

Copyright © 2020 Institute of Advanced Engineering and Science.
All rights reserved.

Corresponding Author:

Muhammad Nadeem,
Department of Computer Engineering,
Faculty of Information and Communication Technology,
Balochistan University of IT, Engineering, and Management Sciences,
Quetta, Balochistan, Pakistan.
Email: dr.nadeem@ieee.org

1. INTRODUCTION

Software development, a rapidly expanding field since the 21st century, the security of the software is a major issue for any organization. Firstly, software applications were limited to a single machine, their maintenance cost and security was also limited. With an increase in an Information Technology infrastructure, applications from single machine moved to multiple machines. Even cloud-based applications were also introduced. Because of the portability and popularity of web-based applications in recent years, desktop applications were replaced by web-based applications in medium and large-scale organizations. On the other hand, web-based applications are less secure than desktop applications. Constructing secure software needs a great deal of security education. Many software developers are not aware of and equipped with enough security education. In addition, many programming books do not teach how to write secure programs [1, 2].

Buffer overflow or Morris Internet worm attack first occurred in 1988. An attacker can easily attack the programs usually developed in unmanaged languages. Buffer overflow happens as a result of poor input validation, causes the system to crash and gets control over program execution. Various manual approaches have been proposed for fixing buffer overflow vulnerability. Manually fixing a vulnerability is a time-consuming task, needs more effort and may induce programming errors. A prototype was developed which uses the mitigation strategies suggested by the public vulnerability repository to fix buffer overflow vulnerability. A static analysis tool was used to search out the weaknesses in the source code. Cybercrimes are constantly increasing. The security of the software is a necessity in this era of fastest-growing software vulnerabilities. A simple error during software development can lead to huge financial losses. Recent research shows that cybercriminals are finding new ways to attack small and large organizations. Kaspersky, a security company reported that a group of cybercriminals intruded more than 100 banks causing a financial loss of more than $1 Billion. As machine learning processes a huge amount of data, hackers target machine learning causing cyberattacks. DDoS attacks are used in IoT based devices by making the whole system down. Physical infrastructure such as media channels, telecommunication networks, hospitals is also the victim of cyberattacks because of low-level security.

According to a Gartner Group report, 70% of the vulnerabilities are found in software applications. For the last 25 years i.e. 1998-2012, Buffer Overflow was the most occurring vulnerability. According to CVE results from 1988-1998, a buffer overflow was declared as the vulnerability of the decade [3]. Common Vulnerabilities and Exposures was launched by MITRE in 1999. It provides a list of identifiers for known software security vulnerabilities. CVE data is synchronized with NVD which immediately updates information regarding fixing vulnerabilities, severity score and impact rating in CVE entries. CVE details provide information about vulnerabilities by their type and their severity level. This information varies year over year. Statistics of buffer overflow reported in CVE are shown in Figure 1, it is obvious that there has been an increase in reported buffer overflow vulnerabilities in recent years.

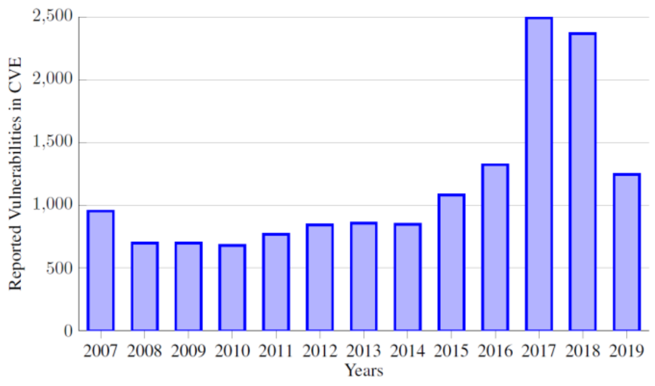


Figure 1. Statistics of buffer overflow in CVE

A Recommender System [4-6], was proposed to achieve higher software security by giving training to software developers. This system uses public vulnerability repository such as CWE, which suggests mitigation strategies for different vulnerabilities. Static analysis techniques were used to search out the vulnerabilities in legacy code. Further, the mapping algorithm was used which mapped the mitigation strategies with the vulnerabilities found. The rest of this paper is organized as follows. Section 2 discusses the background; section 3 discusses the research methodology; section 4 discusses experiments and results, and section 5 gives conclusion and future work.

2. BACKGROUND

Buffer overflow attack [7, 8, 9] can occur if a large amount of data is inserted into a fixed-length buffer than it can occupy and crashes the system by overwriting extra information to an adjacent memory space. Programs written in unmanaged programming languages are usually susceptible as these languages do not have built-in protection against this vulnerability [10]. An attacker takes advantage of this overflow by inserting malicious code into the memory. Different types of buffer overflow along with their existing mitigation strategies are shown in Table 1.

Table 1. Different types of buffer overflows and their mitigation strategies

Types of Buffer Overflows	Proposed Mitigation Strategies
Stack-Based Buffer Overflow	Add boundary check
Heap-Based Buffer Overflow	Add malloc check
Format-String Buffer Overflow	API substitution
Integer-Based Buffer Overflow	Use LONG LONG INT
Unicode Buffer Overflow	Proposed solution doesn’t exist

Stack-Based Buffer Overflow occurs when a string is copied from source buffer to a destination buffer using strcpy function. While copying, strcpy function does not check the size of the destination buffer which may lead to a buffer overflow attack. Vulnerable and healed source code for stack-based buffer overflow is shown in Figure 2, in which array declared as src[] having a size of 12 bytes while the array declared as dest[] having a size of 10 bytes. Instead of the smaller size of the destination array, strcpy function copies the source string to the destination string.

```
/*Vulnerable Code: Stack-Based Buffer Overflow*/
char src[]="Hello World!";
char dest[10];
strcpy(dest,src);

/*Healed Code: Stack-Based Buffer Overflow*/
char src[]="Hello World!";
char dest[10];
strncpy(dest,src,sizeof(dest));
```

(a)

```
/*Vulnerable Code: Heap-Based Buffer Overflow*/
int max=13956798768;
int *ptr=(int*)malloc(max*sizeof(int));

/*Healed Code: Heap-Based Buffer Overflow*/
int max=13956798768;
int *ptr=(int*)malloc(max*sizeof(int));
if(ptr==NULL){
cout<<"Memory is not allocated";
}
else{
cout<<"Memory is allocated";
free(ptr);
```

(b)

Figure 2. Vulnerable and healed code for stack-based buffer overflow, (a) Replacing “strcpy” with “strncpy”, (b) Using “if-else” check

Stack-based buffer overflow can be fixed by adding boundary checks. This strategy makes decisions by checking the size of the source buffer and then comparing it with the destination buffer. In healed code, ‘strncpy’ function was used instead of ‘strcpy’. The difference between the ‘strcpy’ and ‘strncpy’ function is that the strncpy function checks the size of the destination buffer and copy only those bytes till the size of the destination buffer.

Stack-based buffer overflow vulnerability detection technique was proposed in binary codes [11]. Different buffers were scanned to find out the risk functions. Against these risky functions, a function library is established. Finally, vulnerability is detected by comparing the buffer size which is used with the buffer size already declared. The proposed approach only detects the stack overflow vulnerability whereas our approach first detects and then fixes the vulnerability.

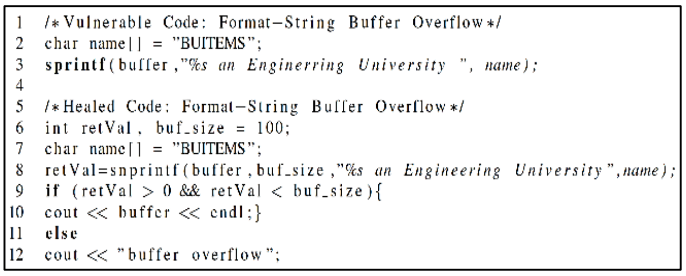
Heap-Based Buffer Overflow. Static memory allocation is used to declare an array of desired space. e.g. In case, we have declared an array to store data for 10 students and if the number of students is changed, then the allocated memory may not fulfill our requirements. To save this memory loss, malloc function was introduced to allocate memory dynamically. An if-else check will be used to check whether a pointer has allocated memory or not?

A smart fuzzing method was used to detect heap-based buffer overflow [12, 13]. Based on concolic execution, this technique detects vulnerabilities more accurately. Addresses of the dynamically allocated memory chunks were stored in a memory table. The memory table is updated upon the allocation and deallocation of these memory chunks. This vulnerability table is then used to create vulnerability constraints. Android-based smart devices are the victims of attackers by inserting malicious code and steal useful information.

Format-String Buffer Overflow occurs by the combination of data and control information. Text strings are sometimes automatically converted to larger formats. sprintf is used to input the formatted string into a character array similarly as ’printf’ is used to write the string into the console. Vulnerable and healed code for format-string buffer overflow are shown in Figure 3(a). While writing the formatted string to the character array, sprintf doesn’t check for the overflow. snprintf is used that checks the length of the buffer and copies a determined number of bytes including the null terminator at the end. An if-else check is added which after checking the condition copies the buffer.

Many approaches such as runtime, testing, and static analysis approaches have been proposed but none of them can distinguish all types of format string vulnerability [14, 15]. The analysis approach finds out the format string functions. Usually, the format string functions are printf, fprintf, sprintf, etc. This approach only detects the vulnerability, however, it reduces the number of false positive.

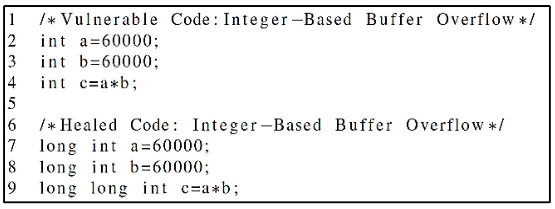
Integer-Based Buffer Overflow occurs when after performing some operation whose value may exceed the maximum or minimum range. e.g. If we add, 100 +200, the result will exceed 8 bits. Integer-based buffer overflow can be fixed by changing the data type from INT which is 32 bit long to LONG LONG INT which is 64 bit long. Vulnerable code for integer-based buffer overflow are shown in Figure 3 (a), whereas the healed code is shown in Figure 3(b).



```
/*Vulnerable Code: Format-String Buffer Overflow*/
char name[] = "BUITEMS";
sprintf(buffer,"%s an Enginerring University ", name);

/*Healed Code: Format-String Buffer Overflow*/
int retVal, buf_size = 100;
char name[] = "BUITEMS";
retVal=snprintf(buffer,buf_size,"%s an Engineering University",name);
if (retVal > 0 && retVal < buf_size){
cout << buffer << endl;}
else
cout << "buffer overflow";
```

(a)



```
/*Vulnerable Code:Integer-Based Buffer Overflow*/
int a=60000;
int b=60000;
int c=a*b;

/*Healed Code: Integer-Based Buffer Overflow*/
long int a=60000;
long int b=60000;
long long int c=a*b;
```

(b)

Figure 3. Vulnerable and healed code for format-string and integer-based buffer overflow, (a) Replacing “sprintf” with “snprintf”, (b) Replacing “INT” with “LONG LONG INT”

The static analysis approach was used for the detection of integer-based buffer overflow by finding sensitive code locations [16]. The static phase performs Control Flow Graph recovery and Call Graph recovery. Once found, an automated POC was generated to fix the vulnerability. This approach was more accurate and doesn’t give any false positive and false negative. IntPatch [17] fixes integer overflow in C/C++ source code at compile time. This approach identifies the un-secure arithmetic operations and fixing statement is inserted after each vulnerable arithmetic operation. A novel approach using symbolic execution was proposed for fixing integer overflow vulnerability in source code [18]. After detecting a vulnerability, a repair pattern was generated which patches the source code. Code refactoring is performed which checks whether the vulnerability is removed or still exists. If a vulnerability exists still there, a message unrepaired integer overflow is generated. This approach was more efficient than manual repair, however, it works only for integer type vulnerability. Unicode Buffer Overflow occurs by injecting Unicode characters into an input that expects ASCII characters. Since ASCII code covers only Western language characters while Unicode can create a character for almost all languages.

Ranges for Unicode characters in different languages are given below:
- English Alphabets (0041-005A, 0061-007A)
- Greek Characters (0370-03FF)
- Mathematical Operators (2200-22FF)
- Arabic Characters (0600-06FF)

If we want to print Alpha Sign having code “nx03B1” in normal mode, we will get a garbage value. Vulnerable and healed code are shown in a listing below. To remove unicode buffer overflow, set the program in Unicode mode and replace cout, printf with wcout, wprintf respectively, the vulnerable and healed code is shown in Figure 4.

```
/*Vulnerable Code: Unicode Buffer Overflow*/
printf(”Alph Sign: \x03B1”);

/*Healed Code: Unicode Buffer Overflow*/
_setmode(_fileno(stdout), _O_U16TEXT);
wprintf(L”Alph Sign: \x03B1”);
```

Figure 4. Replacing ‘printf’ with ‘wprintf’

2.1. Public vulnerability repository

Public software vulnerability repository, Common Weakness, and Enumeration (CWE) hold useful articles about security vulnerabilities and their mitigation strategies. MITRE, launched a CVE list containing a distribution of software weaknesses in 1999. CWE provides a standard list of software vulnerabilities that can be helpful for software developers as well as to large government and private organizations as a knowledge base for these security flaws. These software weaknesses are categorized into three main concepts.

- Research Concepts
- Development Concepts
- Architecture Concepts

These CWE mitigation studies along with existing research studies are combined for developing a prototype tool. This tool detects the vulnerable modules and then fix them as a healed code. Different types of buffer overflow and suggested CWE mitigation strategies are shown in Table 2.

Table 2. Buffer overflow types and CWE mitigation strategies

Types of Buffer Overflows	CWE Mitigation Strategies
Stack-Based Buffer Overflow	Use safer, equivalent functions which check for boundary errors
Heap-Based Buffer Overflow	Perform bounds checking on inputs
Format-String Buffer Overflow	Avoid using functions like ”printf”
Integer-Based Buffer Overflow	Use safe packages such as SafeInt(C++) or IntegerLib(C or C++)
Unicode Buffer Overflow	Use the principle of least privilege

3. EXISTING AUTOMATED APPROACHES

Buffer overflow, a base for many other vulnerabilities. If buffer overflow is handled, more than 50% of the attacks could become ineffective. Vulnerable files were located using tool implementation [19]. Firstly, an application is run under normal condition and a record of the stack trace is made. In the next step, an application with an overflow attack is run again. Results of both the stack traces are compared whether they are similar or different? By tracing the vulnerable path, a vulnerability is removed in an application. However, this tool only works for locating “strcpy” function. Shaw et al. [20] proposed that the legacy C source code can be transformed using program transformation. Safe Library Replacement and Safe Type Replacement were used to heal the vulnerable C language codes. SLR replaces unsafe functions such as strcpy with g strlcpy, memcpy with memcpy s, gets with fgets or gets s, etc. in Linux systems. STR replaces all character pointers with safe data structures i.e., “stralloc” pointers.

The static analysis approach is used for finding buffer overflow vulnerability using tools such as Fortify, Splint, and Checkmarx [21]. Various manual approaches were used for fixing buffer overflow vulnerability. These approaches then guided for the automated fixing of buffer overflow vulnerability. According to [22], web applications were statically analyzed and categorized into dodgy code vulnerabilities, malicious code vulnerabilities, and security code vulnerabilities. These kinds of vulnerabilities are inserted because of developers’ bad programming practices. FindBugs plugin was used to detect vulnerabilities in Java web applications. It references the vulnerabilities to OWASP top 10 and CWE. A secure development framework was developed that should restrict the developers from using bad programming practices. BovInspector, an automated tool, fixes buffer overflow vulnerability in C programs. The tool checks the buffer overflow warning path in a program. These warnings are then validated using symbolic execution. BovInspector fixes these warnings using boundary checks, safer API’s and by extending the buffer size [23].

A software self-healing framework was proposed for the detection and fixing of software security vulnerabilities as shown in Figure 1. Suggested mitigation strategies from CWE were transformed into standardized rules for developing a code transformation module. This module replaced vulnerable code with a healed code. Cross-site scripting or XSS vulnerability was fixed using this prototype. This approach then guided for fixing buffer overflows automatically without human intervention [24]. A combination of both static and dynamic analysis techniques was used for the detection of buffer overflow in binary files [25]. The location of the vulnerability can be found by static analysis and then these vulnerabilities are tested by dynamic analysis to remove false positives. This technique works by converting the binary files into assembly language and then apply both static and dynamic techniques to find the overflow points. Results showed that various overflow functions such as strcpy, strcat and sprintf should be changed by more secure functions such as strncpy, strncat and snprintf respectively. However, our approach uses only static analysis for fixing buffer overflows. An integrated framework [26], was developed to find vulnerabilities in web applications using static analysis approach. The framework is shown in Figure 5. This helps programmers to produce secure code before software development.

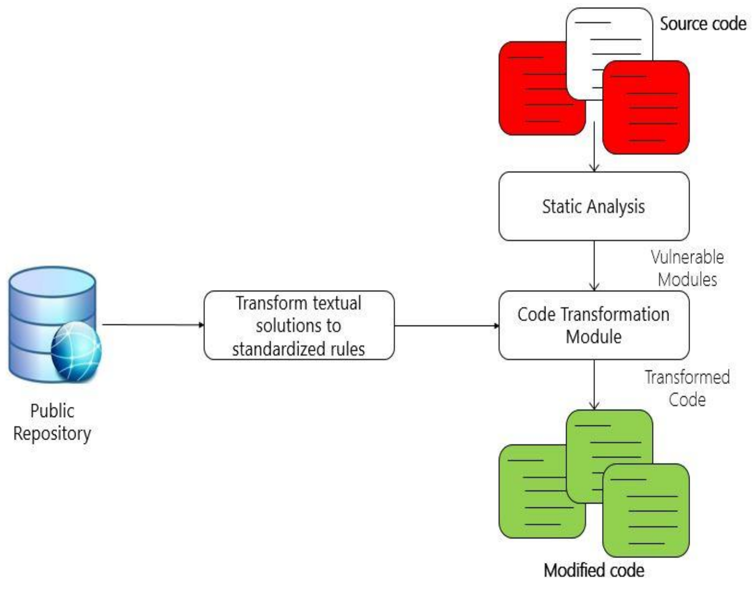


Figure 5. Software self-healing framework [23]

3.1. Proposed automated approach

In our proposed approach, different modules are discussed separately. Table 2 holds useful articles for fixing different types of buffer overflow vulnerabilities. In this work, source code is analyzed using static analysis tool and vulnerable modules are found. Mitigation strategies from public vulnerability repository are transformed into standardized rules. These rules are then used by the source code healing module which transforms vulnerable source code into healed code. If a mitigation strategy against any vulnerability does not exist, then the public vulnerability repository is updated manually. The source code healing module transforms these manual solutions into automated standardized rules resulting in a modified code as shown in Figure 2. Our research comprises of the following research questions.

- RQ1. How can the mitigation strategies in CWE articles for fixing different types of buffer overflow be used to automatically heal the vulnerable source code?
- RQ2. Does the automated approach, in RQ1, compromise the accuracy of the resultant source code?

4. RESEARCH METHODOLOGY

Static analysis is used to analyze source code. Both static and dynamic analysis techniques can be used for finding the buffer overflow vulnerability. However, static analysis shows better results because static analysis checks the vulnerabilities before software deployment. The drawback of this technique is that it results in a high number of false positives. Our proposed system uses mitigation strategies for fixing different types of buffer overflows. These mitigation strategies along with existing research strategies are used to develop a prototype. The solution to fix Unicode buffer overflow was updated manually in a public vulnerability repository. Source code is run through this prototype which replaced vulnerable lines of code with secure lines. A modified version of the self-healing framework is shown in Figure 6.

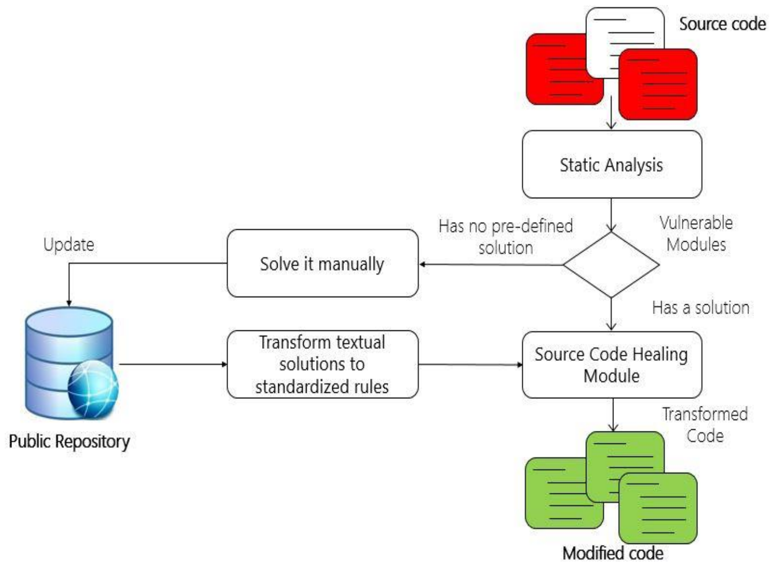


Figure 6. Modified version of software self-healing framework

4.1. Static analysis tool

KIUWAN a static analysis tool scans the source code and measures its quality. Its trial version is free and can be accessible on demand. It references the vulnerabilities defined by OWASP and public vulnerability repositories i.e. CWE. It uses the Code Quality Model for evaluating software characteristics. Some indicators of KIUWAN are shown in Table 3.

Table 3. Summary of kiuwan indicators

Indicator	Description
Security	Repudiate unauthorized users to get access
Reliability	Maintains a specified level of performance
Efficiency	Provides performance relative to available resource
Maintainability	Adaptable to changing requirements and functional specification
Portability	Portability lies in code complexity
Risk Index	Measure how much your code is vulnerable?
Global Indicator	Calculates the weighted average of the characteristics of the application
Defects	Shows how much rules are violated

4.2. Static analysis tool

KIUWAN is a static analysis tool that scans the source code and measures its quality. Its trial version is free and can be accessible on demand. It references the vulnerabilities defined by OWASP and public vulnerability repositories i.e. CWE. It uses the Code Quality Model for evaluating software characteristics. Some indicators of KIUWAN are shown in Table 3.

The methodology of our proposed approach is carried by making the static analysis of vulnerable source code using the KIUWAN static analysis tool. Source code for different types of buffer overflow vulnerability is scanned individually. Results after scanning a source code showed different types of vulnerabilities. These vulnerabilities are further categorized as:

- Vulnerabilities by type
- Vulnerabilities by language
- Vulnerabilities by priority

Among vulnerabilities by type, we selected vulnerabilities related to buffer overflow and among languages C++ was selected. This scanned source code is run through the source code healing module. This module is developed using regular expressions. These regular expressions search for a specific function in source code and the attributes declared with that function were found. Functions such as strcpy, sprintf, etc. were replaced with securer functions like strncpy and snprintf respectively by using regular expressions. Existing research studies for fixing different types of buffer overflow vulnerability along with the CWE mitigation strategies are combined together as a part of this module. The source code healing module replaces vulnerable source code with healed code. Healed code obtained as a result of the source code healing module is then scanned using the same static analysis tool.The methodology used to heal the various types of buffer overflow vulnerabilities is shown in Figure 7. Accuracy of an automated approach will be measured by checking to what extent these vulnerabilities are removed in a healed code. Matrices such as risk index, global indicator, and defects were also compared.

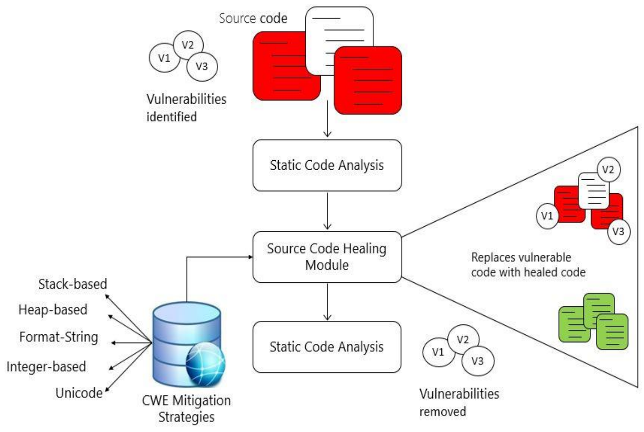


Figure 7. Methodology

5. EXPERIMENTS AND RESULTS

KIUWAN a static analyzer tool was used to detect vulnerabilities in source codes. The experimental setup is shown in Figure 8. Open-source codes for different types of buffer overflow vulnerabilities were scanned individually. A prototype using CWE mitigation strategies to fix different types of buffer overflow vulnerabilities was developed. These codes were passed through a prototype that replaced vulnerable source code with healed code. Healed code was analyzed using the KIUWAN tool. As a result, vulnerabilities are removed. Vulnerable and healed code for stack-based buffer overflow based on the guidelines in CWE121-master are shown in Figure 9.

Vulnerable and healed code for a heap-based buffer overflow is shown in Figure 10. The sample source code can be found at LPT-ArrMinHeap-LeftistTree-Point. Results against different types of buffer overflow vulnerability are shown in Table 4, it is obvious from the results that the proposed prototype tool was able to fix various types of buffer overflow errors in the target source code.

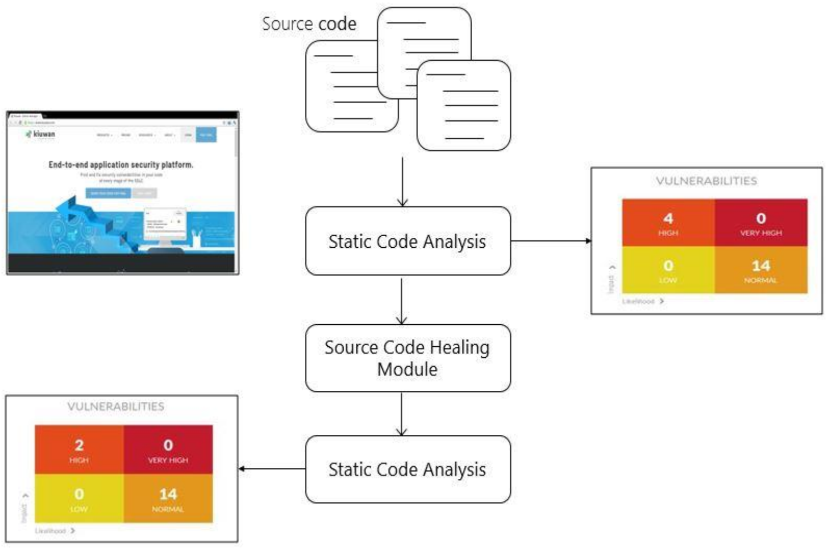


Figure 8. Experimental setup

```
/* Vulnerable Code: CWE121-master.cpp*/
wchar_t dest[50] = L"";
wcscpy(dest, data);

/* Healed Code: New_CWE121-master*/
/*wchar_t dest[50] = L"";*/
wchar_t dest[] = L"";
wcscpy(dest, data);
```

Figure 9. Fixing stack-based buffer overflow

```
/* Vulnerable Code: LPT-ArrMinHeap-LeftistTree-Point*/
bool lefttree::empty(){
return (root == NULL);

/* Healed Code: New_LPT-ArrMinHeap-LeftistTree-Point*/
bool lefttree::empty(){
return (root == NULL);
if(root==NULL){
cout<<"Memory is not allocated";
else{
cout<<"Memory is allocated";
free(root);
```

Figure 10. Fixing heap-based buffer overflow

Table 4. Results: different types of buffer overflows

Buffer overflow type	Target Source Code	No. of identified vulnerabilities	
		Before healing the source code	After healing the source code
Stack-Based Buffer Overflow	CWE121-master	18	16
Heap-Based Buffer Overflow	LPT-ArrMinHeap-LeftistTree-Point	19	5
Format-String Buffer Overflow	Pablodroca/CppUnicodeTests	11	9
Integer-Based Buffer Overflow	CWE190-BigInt-master	11	2
Unicode Buffer Overflow	Test-UNICODE-SYMBOLSr	24	20
	Summary	83	52

It can be observed that 31 buffer overflow errors of various types were fixed. Vulnerable and healed code for a format-string buffer overflow is shown in Figure 11. The sample source code can be found at Pablodroca/CppUnicodeTests. Vulnerable and healed code for integer-based buffer overflow based on the guidelines in CWE 190 is shown in Figure 12. The sample source code can be found at BigInt-master. Vulnerable and healed code for Unicode buffer overflow can be found in Figure 13. The sample source code can be found at Test-UNICODE-SYMBOLS.

```
/*Vulnerable Code: Pablodroca/CppUnicodeTestst*/
sprintf("UTF8 File - PRINTF: %s - LEN: %d\n", unicode_msg, strlen(unicode_msg));

/*Healed Code: New_Pablodroca/CppUnicodeTests*/
int retVal;
//sprintf("UTF8 File - PRINTF: %s - LEN: %d\n", unicode_msg, strlen(unicode_msg));
retVal=snprintf("UTF8 File - PRINTF: %s - LEN: %d\n", unicode_msg, strlen(unicode_msg));
```

Figure 11. Fixing format-string buffer overflow

```
/* Vulnerable Code: CWE190-BigInt-master*/
BigInt a,b,c;
int x;

/* Healed Code: New_CWE190-BigInt-master*/
BigInt a,b,c;
long int x;
```

Figure 12. Fixing integer-based buffer overflow

```
/* Vulnerable Code: Test-UNICODE-SYMBOLS*/
std::cout << unicode_symbols[ch];
std::flush(std::cout);

/* Healed Code: New_Test-UNICODE-SYMBOLS*/
std::wcout << unicode_symbols[ch];
std::flush(std::wcout);
```

Figure 13. Fixing unicode-based buffer overflow

6. CONCLUSION AND FUTURE WORK

In this study, we found that buffer overflow, one of the growing vulnerabilities both in small and largescale software development industries. Manually fixing the vulnerability is a time-consuming task and may induce programming errors. An automated approach was proposed to overcome this problem. A prototype was developed which automatically fixes the buffer overflow vulnerability. Static analysis tool KIUWAN was used for the detection of the vulnerability. Regular expressions were used to replace vulnerable source code with healed code. Results obtained from the resultant source code were more accurate and efficient than the manual fixing of buffer overflow vulnerability. The scope of our study can be extended by focusing on other vulnerabilities such as SQL injection attack, broken access control, etc. Since we have used the CWE vulnerability repository, other public vulnerability repositories can be used as a knowledge base. Experimental results showed that the source code healing module alleviated vulnerabilities against every type of buffer overflow. But, integer-based buffer overflow occupies more memory as INT data type is replaced with LONG LONG INT. In the future, the identified drawback of this study can be minimized.

REFERENCES

[1] A. Agrawal, M. Alenezi, R. Kumar, and R. A. Khan, “A source code perspective framework to produce secure web applications,” *Computer Fraud & Security*, vol. 2019, no. 10, pp. 11–18, 2019.

[2] H. Ashraf, M. Alenezi, M. Nadeem, and Y. Javid, “Security assessment framework for educational erp systems,” *International Journal of Electrical and Computer Engineering (IJECE)*, vol. 9, no. 6, pp. 5570–5585, 2019.

[3] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole, “Buffer overflows: Attacks and defenses for the Vulnerability of the decade,” in *Proceedings DARPA Information Survivability Conference and Exposition, DISCEX’00, IEEE*, vol. 2, pp. 119-129, 2000.

[4] M. Nadeem, E. B. Allen, and B. J. Williams, “A method for recommending computer-security training for software developers: Leveraging the power of static analysis techniques and vulnerability repositories,” in *2015 12th International Conference on Information Technology-New Generations. IEEE*, pp. 534–539, 2015.

[5] M. Nadeem, B.J. Williams, G.L. Bradshaw, and E.B. Allen, “Human subject evaluation of computersecurity training recommender,” in *IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, pp. 251–256, 2016.

[6] V. Raghavan and X. Zhang, “An integrative model of managing software security during information systems development,” *Journal of International Technology and Information Management*, vol. 26, no. 4, pp. 83-109, 2017.

[7] S. Nashimoto, *et al.*, “Intrusion prevention system inspired immune systems,” *Indonesian Journal of Electrical Engineering and Computer Science (IJEECS)*, vol. 2, no. 1, pp. 168–179, 2016.

[8] S. Nashimoto, *et al.*, “Buffer overflow attack with multiple fault injection and a proven countermeasure,” *Journal of Cryptographic Engineering*, vol. 7, no. 1, pp. 35–46, 2017.

[9] J. Ren, Z. Zheng, Q. Liu, Z. Wei, and H. Yan, “A buffer overflow prediction approach based on software metrics and machine learning,” *Security and Communication Networks*, vol. 2019, 2019.

[10] N. McKelvey, “Developing a secure programming module to cope with modern vulnerabilities,” *International Journal of Information and Network Security*, vol. 1, no. 1, paper. 41, 2012.

[11] W. Xie, J. Hu, P. K. Kudjo, L. Yu, and Z. Zeng, “A new detection method for stack overflow vulnerability based on component binary code for third-party component,” in *2018 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation. IEEE*, pp. 1255–1259, 2018.

[12] M. Mouzarani, B. Sadeghiyan, and M. Zolfaghari, “A smart fuzzing method for detecting heap-based buffer overflow in executable codes,” in *2015 IEEE 21st Pacific Rim International Symposium on Dependable Computing (PRDC), IEEE*, pp. 42–49, 2015.

[13] M. Mouzarani, B. Sadeghiyan, and M. Zolfaghari, “Smart fuzzing method for detecting stack-based buffer overflow in binary codes,” *IET Software*, vol. 10, no. 4, pp. 96–107, 2016.

[14] W. Han, M. Ren, S. Tian, L. Ding, and Y. He, “Static analysis of format string vulnerabilities,” in *2011 First ACIS International Symposium on Software and Network Engineering. IEEE*, pp. 122–127, 2011.

[15] M. J. Khalsan and M. O. Agyeman, “An overview of prevention/mitigation against memory corruption attack,” in *Proceedings of the 2nd International Symposium on Computer Science and Intelligent Control*, ACM, paper. 42, 2018.

[16] B. Zhang, C. Feng, B. Wu, and C. Tang, “Detecting integer overflow in windows binary executables based on symbolic execution,” in *2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), IEEE*, pp. 385–390, 2016.

[17] C. Zhang, *et al.*, “Intpatch: Automatically fix integer-overflow-tobuffer- overflow vulnerability at compile-time,” in *European Symposium on Research in Computer Security. Springer*, pp. 71–86, 2010.

[18] P. Muntean, M. Monperrus, H. Sun, J. Grossklags, and C. Eckert, “Intrepair: Informed fixing of integer overflows,” *arXiv preprint arXiv*: 1807.05092, 2018.

[19] A. Iyer and L. M. Liebrock, “Vulnerability scanning for buffer overflow,” in *International Conference on Information Technology: Coding and Computing, 2004. Proceedings, ITCC 2004*, vol. 2, pp. 116–117, 2004.

[20] A. Shaw, D. Doggett, and M. Hafiz, “Automatically fixing c buffer overflows using program transformations,” in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 124–135, 2014.

[21] T. Ye, L. Zhang, L. Wang, and X. Li, "An empirical study on detecting and fixing buffer overflow bugs," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST), IEEE,* pp. 91–101, 2016
[22] M. Alenezi and Y. Javed, "Open source web application security: A static analysis approach," in *2016 International Conference on Engineering & MIS (ICEMIS), IEEE*, pp. 1–5, 2016.
[23] F. Gao, L. Wang, X. Li, "Bovinspector: automatic inspection and repair of buffer overflow vulnerabilities," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE*, pp. 786–791, 2016.
[24] A. Ur, R. Jaffar, M. Nadeem, M. Alenezi, and Y. Javed, "Using public vulnerabilities data to self-heal security issues in software systems," *ICIC Express Letters*, vol. 13, pp. 557–567, 2019.
[25] J. Yuan and S. Ding, "A method for detecting buffer overflow vulnerabilities," in *2011 IEEE 3rd International Conference on Communication Software and Networks*, pp. 188–192, 2011.
[26] M. Alenezi and Y. Javed, "Developer companion: A framework to produce secure web applications," *International Journal of Computer Science and Information Security*, vol. 14, pp. 12–16, 2016.

BIOGRAPHIES OF AUTHORS

Aamir Shahab holds a graduate degree in Computer Engineering from BUITEMS, Quetta (2019). He obtained an undergraduate degree in Computer Engineering from Bahauddin Zakariya University, Multan, Pakistan (2016). He is currently pursuing his Ph.D. program in Computer Engineering with an emphasis on cybersecurity. He is serving as a visiting faculty member at BUITEMS, Quetta. His researches are in the field of software security. He is affiliated with IEEE as a student member. Besides, he is interested in Cyber Security research and training.

Mamdouh Alenezi is currently the Dean of Educational Services at Prince Sultan University. Alenezi received his MS and Ph.D. degrees from DePaul University and North Dakota State University in 2011 and 2014, respectively. Alenezi has published several journals and conference papers about software security. He has extensive experience in data mining and machine learning where he applied several data mining techniques to solve several Software Engineering problems. He conducted several research areas and development of predictive models using machine learning to predict fault-prone classes, comprehend source code, and predict the appropriate developer to be assigned to a newly reported bug.

Muhammad Nadeem holds a doctorate degree in Computer Science from Mississippi State University, United States. His field of specialization is Software Engineering with a specific focus on Software Security. He is Fulbright alumnus and has been honored with Best University Teacher Award byes and has also served as member organizing committee for IEEE ICE Cube Conference. His skills include software development, automated static code analysis, vulnerability identification, and mitigation of vulnerabilities based on industry best practices. He has been leading an in-house software development teams in public sector organizations. He has also been rendering consultancy services in the implementation of Oracle and PeopleSoft products in different public sector organizations. He has presented his research on software security in several international ACM and IEEE conferences in the United States. He is a professional member of IEEE and ACM. He is also a reviewer for two international conferencnizations. He has also rendered services in academia, including teaching in the United States, and has held different administrative positions.

Raja Asif Wagan was born in Jamshoro, Sindh Province, Pakistan in 1983. He received his bachelor's degree in computer science in 2005 from the University of Sindh, Pakistan. Furthermore, he received his Masters of Information Technology from Universiti Utara Malaysia. Now he is pursuing his Ph.D. in information and communication engineering at Harbin Engineering University, Harbin, China. His research interests concentrate on routing protocols, wireless sensor networks, and computer networks.