STATIC VULNERABILITY PATTERN DETECTION IN LOW LEVEL PROGRAMMING LANGUAGE

Mansour Al-Qattan/PhD Candidate,
Feng Chen/Senior Lecturer
Software Technology Research Laboratory,
De Montfort University, Leicester - UK

ABSTRACT

Vulnerability checking tools in the software industry mostly focus on high-level programming languages, and vulnerability detection in low-level languages, unfortunately, has been largely sidelined in the case of legacy systems. This research proposes a method for finding vulnerabilities in an assembly language through wide-spectrum language (WSL) with FermaT using the static tainted vulnerability analysis technique with the slicing transformation FermaT engine. Our method decompiles the binary executable file to assembly and translates the assembly to WSL, and then detects vulnerabilities by combining the FermaT slicing transformation with taint analysis. The results show that WSL FermaT can detect vulnerability in a binary executable file easily as FermaT contains multiple transformations that enable developers to meet their requirements.

Keywords: *vulnerabilities, vulnerability detection, static analysis, program transformation, FermaT, wide-spectrum language.*

INTRODUCTION

Nowadays, many companies are attempting to transform legacy systems to higher levels to be compatible with new technology. However, these programs were developed when security was absent and programs based on assembly were very difficult to track for vulnerability. It is very costly as there is a lack of experience with assembly and the endeavour is time consuming, further increasing the associated financial burden to hire professional people. Over the years, the vulnerabilities posed by the gaps and loopholes had the possibility of being abused by malicious attacks to intrude upon the security mechanism. According to CERT and NVD (2016), these vulnerabilities still exist.

Software systems are developing at a rapid rate in terms of size and complexity, leading to a serious increase in the number of bugs. Such increasing trends in bugs are causing a serious threat for the software industry in the form of overcoming security vulnerability challenges. One of the most known and exploitable vulnerabilities throughout the evolution of the software industry is buffer overflow (BOF) because even the infamous “Heartbleed” vulnerability is a memory BOF. These BOFs allow the program to crash or execute random codes [3, 14]. This vulnerability has gained serious attention from researchers in terms of designing and developing tools and techniques for mitigating, i.e., testing [26], monitoring [12], patching [19] and fixing [5]. Irrespective of the various methods and tools, these vulnerabilities are still identified in various legacy systems and in the latest programs [21].

Most security attacks, in general, target exploiting the weaknesses in source code. The techniques to detect the vulnerabilities, therefore, must be integrated in the software development process. Information from the National Vulnerability Database (NVD) for the historical period, reveal the proportional increase in vulnerabilities with respect to software size and its complexity. These reports also uncover the share of BOF vulnerabilities (approximately 10% from 2010–2011, see Fig. 2) and their impact on the software industry. Finding related coding faults via developing a static implementation to recognize the semantics of WSL code and identifying vulnerabilities is the purpose of this work. Static analysis and related tools assist to free a program from BOF vulnerabilities and various techniques in this research were developed to optimize the code using FermaT with WSL. Currently, a number of static analysis methods in the literature are being introduced to identify vulnerabilities because of widespread security breaches in software programming [7, 10, 20]. It is agreed [4, 27] that characterizing such weaknesses and loopholes to facilitate the further rectification is the basic purpose of vulnerability analysis. Several evaluations turn out to be effective in detecting BOF vulnerabilities based on various benchmark programs [16, 31] using static analysis-based tools. To select a particular method for identifying BOF vulnerabilities, no appropriate detection methods are defined or confirmed because of a lack of comprehensive comparative studies.

A novel static vulnerability analysis approach to detect BOF vulnerable programs in a low-level language has been introduced in this work. The present research focuses on improvising vulnerability detection analysis using WSL because of its excellent capacity to reverse engineer low-level languages and this helps analyze the severity of the vulnerability of legacy systems during migration. Our technique is two-fold: 1) finding the vulnerability by combining the slicing transformation with taint analysis; and 2) boundary checking for the value range size.

The proposed methodology does not simply scan for vulnerability; however, it also essentially utilizes the transformations to help identifying the threats of the source code accurately, though this is preliminary in nature and uses FermaT slicing transformation to improve the analysis method of vulnerability detection when migrated from assembly, which enables us to find the problems in the assembly code.

RELATED WORK

Various static analysis tools discover security weaknesses when the program is not executing by scanning the program code based on potential assumptions. Overall, the goal of these static tools includes extracting and assessing the information of a given source code. In recent times, these tools have also been utilized to visualize the source code of a given program [12]. Researchers have carried out a significant number of studies seeking to examine static vulnerabilities of different level language applications. Early versions of languages, such as C and C++, did not deliver built-in defence against issues similar to BOF based on a lack of bounds-checking mechanism. Examining and rectifying problems with accessing and/or overwriting parts of memory using built-in buffer types, such as arrays (BOF), were a matter of importance. The BOF phenomenon allowed access to parts of memory that made applications vulnerable to security breaches and resulted in the unexpected. Such vulnerabilities have been intentionally exploited by hackers [30]. Most of the injections (around 55%)

are of BOFs and are dealt with successfully by utilizing paradigm slicing approaches. The Syntax Oriented Tools category is comprised of tools that employ and contain lexical analyzers and pattern matching. ITS4 [20] and RATS [10], for example, follow this kind of approach.

There are many other tools helpful for discovering various security vulnerabilities or to eliminate false positives. In [22], the basic focus was on analyzing security vulnerability detection of BOFs; however, [23] employed a vulnerability database similar to ITS4 and RATS. In such scenarios, each tool has its own way of presenting the results. For example, SSVChecker provides a way to exploit differences of each analysis report from each tool. Typical static analysis security vulnerability detection tools document potential security vulnerabilities in a format not easily read by software developers. Code annotations are typically used to indicate semantic properties of a program source code. These are also considered to be additional syntax constructs that aid a programmer in discovering the vulnerable areas in the code through annotation [9]. Splint is a well-known annotating tool utilized extensively to this end [8]. [29] put forth the necessity to deal with vulnerability patterns at the development stage compared to post-development issues. These authors' highlighted two main reasons for such urgency: 1) the issues are expensive to understand and mange if they are not dealt with early; and 2) developers become ignorant to the vulnerabilities once the complete program is developed.

Various insecure ANSI C library pointers, function calls, pointer arithmetic, arbitrary buffer indexes, etc. are responsible for BOF vulnerabilities. Some of these tasks explicitly limit the detection process, analysis and scope. For example, most BOF vulnerabilities are because of library function calls with unsanitized arguments in a C program [32]. Similarly, Dor et al. observed BOF vulnerabilities based on string variables [7]. In terms of the majority of BOF vulnerabilities, string pattern matching is very common [20, 33, 34] and to the result of function calls and arguments. Evans et al. discovered BOF vulnerabilities stemming from manipulator cleared functions, where annotations are delivered properly [9]. Most of the pointer arithmetic and condition expressions in a branch or loop are linear [25].

FERMAT AND WIDE-SPECTRUM LANGUAGE (WSL)

The transformation engine available in FermaT is beneficial for transforming legacy assembler systems and their applications for present business operations. FermaT possesses various tools for assembler documentation, transformations, and migration of a program. The latest version of FermaT tools even helps build the assembler code documentation, functional logic, data analysis, identification of business rules, and to migrate code from one language to other.

Formally proven programming transformations are included in FermaT to refine or preserve semantics of a program at the time of changing a program from one form to another. Transformations in FermaT aid restructuring and simplifying the existing legacy systems and extracting the information into equivalent high-level language representations. The originality of the code logic is not affected during the process of restructuring and transforming. A large catalogue included with proven transformations along with verified mechanical applicable conditions are available in WSL. The other main feature of the FermaT migration engine is having all mathematically-proven transformations preserve the semantics of a subjected program.

WSL is designed to support both low-level and high-level languages and is possibly considered to be a non-executable specification language. Such languages are designed for supporting a defined program methodology based on program refinement. The advantage of wide-spectrum language is to be incrementally refined with other intermediate versions of other high- or low-level language constructs.

PREPOSED METHOD AND ANALYSIS ENVIRONMENT

A. Information of a Program at the Semantic Level

Translation: The investigation plan for our research was considered and translated existing vulnerabilities code in C language into the following (binary execution file of C code and then to assembly) WSL in such a way that certain classes of vulnerabilities translate to specific semantic behaviours in the generated WSL code. Therefore, the vulnerabilities in the existing code can be determined by analysis of the WSL. High-/low-level modelling of the program is required. We modelled buffer overflows in WSL to define a single array or sequence which models the memory of the C program. In this model, a C variable, string, or array is represented by subsequence in the memory model.

B. FermaT and Proposed Method in Vulnerability Detection

The FermaT transformation engine was employed after translation and rules were established to detect possible buffer overflow on a string or array which checks the array boundaries and reports on the program with a notification if the boundaries are exceeded. This program is semantically equivalent to the original for the initial states in which no buffer overflow occurs; however, this program changes the semantics upon buffer overflow to those easier to detect.

Vulnerability analysis: the vulnerability detection tool in WSL does not currently have many useful applications. So, a taint analysis tool for WSL was established to aid determining potential vulnerabilities to trace these functions, assignments or variables with a FermaT transformation rule, slicing, in order to isolate and extract the relative nodes.

Taint analysis: the unsafe inputs are found and assigned as tainted. This tainted data is calculated using a fixed-point algorithm. If such tainted programs are employed in sensitive operations, the relevant statements are flagged as vulnerable programs.

Slicing transformation method: a backwards/forwards conditioned semantic slice of the program from the buffer assignment automatically eliminates codes not relevant to the particular assignment and reduces the amount of work needed to either prove the non-existence/existence of buffer overflow or compute an example input state and control flow path which may result in buffer overflow. Then, the vulnerability is identified with a different set of tools that can be applied to the program to inspect for vulnerabilities (for example, using dataflow analysis). In this way, many slightly different variations of the same vulnerability can be detected as they are transformed to the same code.

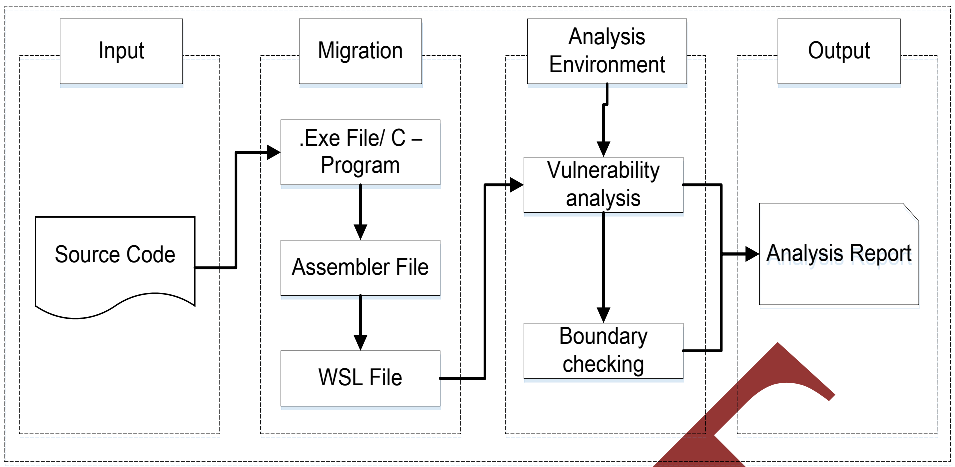


Figure 1: Detection Process Framework

Program slicing patterns generally involve the three following steps: 1) Automatic decomposition of program; 2) Data analysis; and 3) A control flow. Inter-procedural slicing with dependence graphs is included with System Dependence Graph (SDG) [11]. The slicing was based on various program dependency graphs (PDGs), useful for dependency modelling. This includes a main procedure along with auxiliary procedures using the C language as in Figure 1.

In the program shown in Figure 4, there are three instances of slicing variables x, y and z that remain slice variables with relevant line even when the rest of the program is changed. A slicing program is a subset of the main program and the changes in the sliced program do not affect the originality of the main code program, and the results of the total program will be the same.

```
// Simple Program
Void Slicing ()       1
{                     2
  int x=0;            3
  int y=0;            4
  int z=6;            5
    If (z>15)         6
      x++;            7
    Else              8
      y++;            9
}                     10
```

```
// Slice for Variable x in
7th Expression
Void Slicing ()       1
{                     2
  int x=0;            3
  int z=6;            5
    If (z>15)         6
      x++;            7
 }                    10
```

```
// Slice for Variable z in
6th Expression
Void Slicing ()       1
{                     2
  int z=6;            5
}                     10
```

```
// Slice for Variable y in
9th Expression
Void Slicing ()       1
{                     2
  int y=0;            4
  int z=6;            5
    If (z>15)         6
      Else            8
        y++;          9
}                     10
```

Figure 2. Slicing Program to Identify Vulnerabilities

Boundary checking analysis: the only way to check for vulnerabilities is by modelling the other language behaviour. For example, it can occur for BOF by making the index in an array explicitly behave as a memory location and then start to add values inside the array as stack memory. Next, one can verify the local variable if the code saves the boundaries, since it can be proven that the assembly and C code have been translated to WSL and has vulnerabilities. Additionally, a method for the global variable was employed and saved space based on the previous size of the variable that had been overwritten. Therefore, the method was verified and started placing the global variable in the index and if it exceeded that size, we assumed a potential vulnerability. The memory was analysed in terms of stack and heap requirements.

Regarding a stack, the space is managed efficiently by an array, and there are only local variables and there is a limit on stack size. With respect to the heap, globally, the variables are available, and there is not any limit for the size of memory, for that memory can be manageable, a user can allocate and free the variables, and using realloc() can resize the variables is possible.

The value range was analysed based on the above discussion as well as the range of sensitive variables ahead of defining the conditions for BOF weaknesses. The value range analysis incorporated the program’s lower and upper bounds of certain variables. Such facts are used at the time of calling a function.

Output

We compared the end output results of the C2WSL and ASM2WSL to establish differences and then compared the detecting result from the boundary and vulnerability analysis in order to make a decision as a report analysis.

VUNERABILITY RULES AND VULNERABILITY DETECTION IMPLEMENTATION

Vulnerabilities are usually correlated to valid codes that follow the goals of the programmers according to the compiler point of view. However, this is a weakness and each vulnerability pattern consists of numerous fundamentals that are a part of the C language. Certain specific sets of elements in the C language behave as vulnerability patterns, so by translating to assembly and WSL, they show more details in WSL. In software language terminology, such elements are usually called “tokens”. By detecting and examining the tokens responsible for BOFs, one can recognize vulnerable patterns. Vulnerable attributes (VA) is a new term for holding relevant information on tokens. VAs contain type, size, scope, etc. and are different from tokens. To detect vulnerabilities of source code, semantic level information of parts of the code is extracted.

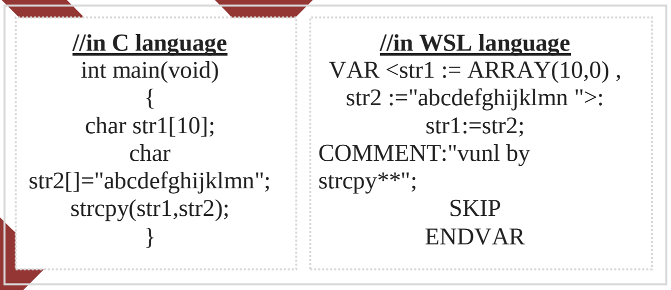
Table 1: VRs Useful for Detecting Vulnerable Code

C Vulnerability Pattern	Vulnerability Rule(VR)
strcpy(dst_Var,src_Var);	Src_VarSize ≤ dst_VarMaxSize
strncpy(dst_Var,src_Var,n);	Min(src_VarSize,n) ≤ dst_VarMaxSize

x="Overwritten";	Size("Overwritten") ≤ xMaxSize
strcat(x,Buff);	x.Size + Buff.Size ≤ x.MaxSize - 1
memcpy(dst_Var,src_Var,n);	Min(src_Var.size,n) ≤ dst_Var.MaxSize
gets(x);	∞ ≤ x.Size
strncat(x,Buff,n);	x.Size + min(Buff.size,n) ≤ x.MaxSize - 1

Every given function in Table 1 has some kind of restriction. In VRs, formal constraint language is utilized and the vulnerability rules implemented in WSL must be in a position to parse and analyze programs correctly. The analysis tool for the WSL program must have at least one parser compatible with other compilers. However, lexical analysis tools are confusing based on similar names utilized for vulnerable functions. In this case, the abstract syntax tree (AST) analysis differentiates identifiers.

For example, when considering the value "size" of a variable characteristic, the size of the first variable is stored in "size" with a sign corresponding node, “a”. In the following node, the size of the second variable is assumed to be “b" which is known. In this way, traversing the AST clarifies the VAs of the program. During the following phase, the extracted values are stored in nodes utilized to detect vulnerabilities. As each vulnerability is specified as a set of VRs on VAs, it is possible to determine vulnerabilities by checking the compatibility of every rule. In this step, the AST is traversed again and the VRs are applied on each of its nodes. If one of these rules is broken, the matching node is assigned as a vulnerable node. The following rule is applied: VR| a.Size + b.Size ≤ b.MaxSize − 1 | False Rule = vulnerability detected. In the WSL code in Figure 3, it contains different sections made up of basic blocks including a single entry and exit point at both the top and bottom ends, respectively. These blocks contain a CFG that is useful to define control dependencies for the given program.



```
//in C language
int main(void)
{
char str1[10];
char
str2[]="abcdefghijklmn";
strcpy(str1,str2);
}
```

```
//in WSL language
VAR <str1 := ARRAY(10,0) ,
str2 :="abcdefghijklmn ">:
str1:=str2;
COMMENT:"vunl by
strcpy**";
SKIP
ENDVAR
```

Figure 3. Equivalent C Program - WSL Translation from ASM2WSL with a BOF Vulnerability

A FermaT transformation tool was employed on the code in Figure 4 for translating from assembly to WSL, as depicted in Figure 3. Translation from assembly to WSL produces the action system and in order to restructure the action system, the transformation engine generated the structured code in WSL.

In Figs. 2 and 3, the BOF vulnerability for strcpy() demonstrates overwriting taking on both sets of code. The incorrect usage of strcpy() and strcat() is responsible for such overwriting. In particular, the BOF takes place when cpy or copy has a great amount of data.

Figure 4 also shows a vulnerability pattern as this is clarified with extracting security vulnerabilities in the source code. In the previous example, the associated syntax pattern is: Variable.size() (parameter1.size(), parameter2.size()); If the size of parameter2 becomes larger than parameter1, a BOF occurs. There is clearly a relationship between the variables, and these relationships form another aspect of VRs.

The buffers are modelled as couples of value ranges by tracing the contents of string variables using a slicing method. Each string is modelled with two attributes: 1) the number of bytes allocated in the string buffer (i.e. maxsize) and 2) the number bytes utilized at the present state. As a consequence of the size and length of the strings, most of the C library functions can be modelled irrespective of string content. Generally, VRs are considered to be a set of numeric constraints which in general are applied on a parse tree of a program. The functions working with buffers are tested to identify the defined constraints. If they match with the defined constraints, then a report warning of BOF vulnerabilities are generated with the corresponding VR.

//in Assembly language	**//in WSL language**

Mov DWORD PTR [esp+23], 1684234849
Mov DWORD PTR [esp+27], 1751606885
Mov DWORD PTR [esp+31], 1818978921
Mov WORD PTR [esp+35], 28269
Mov BYTE PTR [esp+37], 0
Lea eax, [esp+23]
Mov DWORD PTR [esp+4], eax
Lea eax, [esp+38]
Mov DWORD PTR [esp], eax
	stack[1] := "a"; stack[2] := "b";
	stack[3] := "c"; stack[4] := "d";
	stack[5] := "e"; stack[6] := "f";
	stack[7] := "g"; stack[8] := "h";
	stack[9] := "i"; stack[10] := "j";
	stack[11] := "k"; stack[12] := "l";
	stack[13] := "m";
	stack[14] := "n";
	eax := stack[1..14];
	eax := SLENGHT(eax)

Figure 4. Equivalent Assembly Program in WSL

VALIDATION AND EVALUATION

The vulnerability detection tool proposed in this research was tested on a variety of sample programs to investigate the relative effectiveness, vulnerability and boundary checking analysis and combination of both the techniques. This research includes three sets of tests: 1) displaying the boundary checking analysis algorithm and the given sample program analyzed based on the value range method; 2) different sets of tests conducted to display vulnerabilities using the combined form of taint analysis and slicing; and 3) the repetition of both steps 1 and 2 to test if both algorithms are enabled. The vulnerability analysis mode tries to verify whether the tool was capable of detecting all manners of vulnerabilities in the given sample program or not. The demonstration of precision and recall for validations is as follows:

% of functions that are classified correctly:

$$\boldsymbol{Accuracy} = \frac{\boldsymbol{TN\ + TP}}{\boldsymbol{TN + FN + FP + TP}}$$

Probability of detecting a vulnerability (recall) =

$$\boldsymbol{PD} = \frac{\boldsymbol{TP}}{\boldsymbol{TP + FN}}$$

Probability of misclassifying a good function =

$$\boldsymbol{PF} = \frac{\boldsymbol{FP}}{\boldsymbol{TN + FP}}$$

As a bad function (false alarm), how close is the result to the

$$\boldsymbol{Balance} = 1 - \frac{\sqrt{(\boldsymbol{0-PF})^{2} + (\boldsymbol{1-PD})^{2}}}{\sqrt{\boldsymbol{2}}}$$

Ideal point (pf, pd) = (0, 1)

The addition of our tool improves the accuracy of the vulnerability detection, minimizing the number of false positives. Now the predicated values can be compared with actual values and can determine the count of correct predications. This in turn gives us a measure for precision and recall (see Figure 5):

Precision: a measure of predicted vulnerability components accurately. Lower the number of false positives means higher the precision. The proposed detection tool is efficient enough to predict vulnerabilities correctly.

Recall: a measure of the potential vulnerable components essentially predicted. High recall means low false negatives.

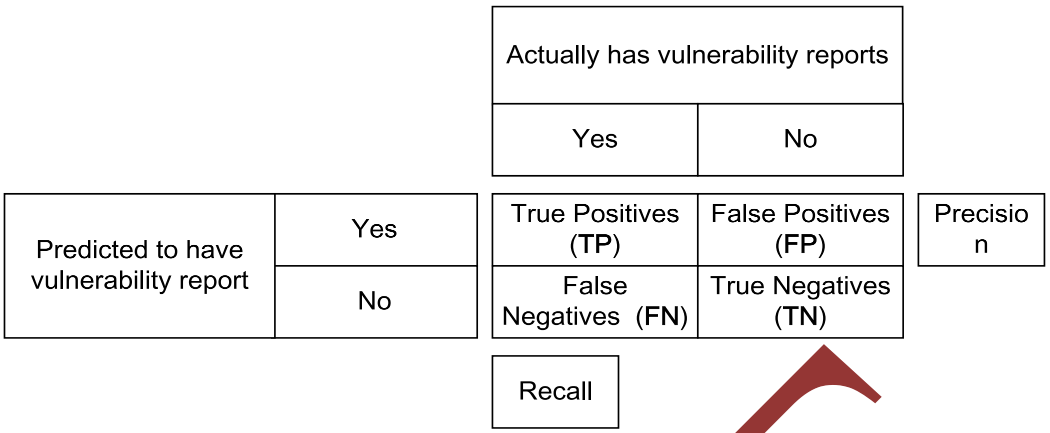


Figure 5: Recall and Precision Clarified. Recall is TP/(TP + FN); Precision is TP/(TP + FP)

To test the effectiveness of proposed system, two competing vulnerability detection tools were tested and compared with the results obtained by the proposed WVD tool. Flawfinder [23] and SPLINT [8] have been employed extensively for security. Flawfinder uses a pattern matching approach for detecting vulnerabilities while SPLINT utilizes annotation-based data-flow analysis to detect vulnerabilities. The results obtained by WVD are tested and compared with the results obtained by Flawfinder and SPLINT.

CONCLUSION AND FUTURE WORK

Unconscious errors committed by programmers seem to be the primary origin of the majority of source code vulnerabilities. Hence, a new method was proposed for detecting vulnerabilities using a static detection method at the semantic level for BOFs. Favourable results were obtained while detecting vulnerabilities in an assembly program with WSL code. It is important to use an optimization method in a formal language with a vulnerability detector specific for memory leak problems, very common in most programming languages. The FermaT transformation engine has many advantages for building tools based on such requirements. The variety of transformations help in so many ways, i.e. to demonstrate the correctness of a program's implementation for the given specifications, to transform a low-level assembler language program to a more comprehensible high-level language program and even to abstract specifications. Thus, the combination of these approaches to detect potential vulnerabilities using a fusion of taint analysis and FermaT's transformations in WSL presents a powerful method for detecting potential vulnerabilities. The most evident advantage of this work lies in its ability to detect and analyze vulnerabilities in an effective way at the level of a binary executable file.

REFERENCES

[1] M. Akbari, S. Berenji and R. Azmi , "Vulnerability detector using parse tree annotation," In Education technology and computer (ICETC), 2010 2nd international conference on, 2010, pp. V4-257-V4-261.

[2] [2] A. Atkins, N. Reznikov, L. Ofer, A. Masic, S. Weiner and R. Shahar, "The three-dimensional structure of anosteocytic lamellated bone of fish," Acta biomaterialia, vol 13, pp. 311–323, 2015.

[3] [3] B. Chess and G. McGraw, "Static analysis for security," IEEE security & privacy, no 6, pp. 76–79, 2004.

[4] [4] M. Cova, V. Felmetsger, G. Banks and G. Vigna, , "Static detection of vulnerabilities in x86 executables," In 2006, pp. 269–278.

[5] [5] C. Dahn and S. Mancoridis, "Using program transformation to secure C programs against buffer overflows," In 2003, pp. 323.

[6] [6] J. Dehlinger, Q. Feng and L. Hu, "Ssvchecker: Unifying static security vulnerability detection tools in an eclipse plug-in," In Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange, 2006, pp. 30–34.

[7] [7] N. Dor, M. Rodeh and M. Sagiv, "CSSV: Towards a realistic tool for statically detecting all buffer overflows in C," In ACM sigplan notices, 2003, pp. 155–167.

[8] [8] D. Evans, "Splint home page ", [Online] [Accessed 6/9/2016].

[9] [9] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis," Software, IEEE, vol 19, no 1, pp. 42–51, 2002.

[10] [10] B. Hackett, M. Das, D. Wang and Z. Yang, , "Modular checking for buffer overflows in the large," In Proceedings of the 28th international conference on software engineering, 2006, pp. 232–241.

[11] [11] S. Horwitz, T. Reps and D. Binkley, "Interprocedural slicing using dependence graphs," ACM transactions on programming languages and systems (TOPLAS), vol 12, no 1, pp. 26–60, 1990.

[12] [12] R.W. Jones and P.H. Kelly, "Backwards-compatible bounds checking for arrays and pointers in C programs." In Aadebug, 1997, pp. 13–26.

[13] [13] S. Neuhaus, T. Zimmermann, C. Holler and A. Zeller, , "Predicting vulnerable software components," In Proceedings of the 14th ACM conference on computer and communications security, 2007, pp. 529–540.

[14] [14] A. One, "Smashing the stack for fun and profit," Phrack magazine, vol 7, no 49, pp. 14–16, 1996.

[15] [15] G. Paul, "7. memory : Stack vs heap ", [Online] [Accessed 6/9/2016].

[16] [16] D. Pozza, R. Sisto, L. Durante and A. Valenzano, "Comparing lexical analysis tools for buffer overflow detection in network software," In Communication system software and middleware, 2006. comsware 2006. first international conference on, 2006, pp. 1–7.

[17] [17] L. V. SATYANARAYANA and M. C. SEKHAR, "Static analysis tool for detecting web application vulnerabilities," .

[18] [18] H. Shahriar and M. Zulkernine, , "Classification of static analysis-based buffer overflow detectors," In 2010 fourth international conference on secure software integration and reliability improvement companion, 2010, pp. 94–101.

[19] [19] A. Smirnov and T. Chiueh, "Automatic patch generation for buffer overflow attacks," In Information assurance and security, 2007. IAS 2007. third international symposium on, 2007, pp. 165–170.

[20] [20] J. Viega, J. Bloch, T. Kohno and G. McGraw, "Token-based scanning of source code for security problems," ACM transactions on information and system security (TISSEC), vol 5, no 3, pp. 238–261, 2002.

[21] [21] C. Vulnerabilities, Common vulnerabilities and exposures, 2005.

[22] [22] D.B. Wagner, "Buffer overrun detection", [Online] [Accessed 6/9/2016].

[23] [23] D.A. Wheeler, "Flawfinder home page ", [Online] [Accessed 6/9/2016].

[24] [24] J. Wilander, "Contributions to specification, implementation, and execution of secure software," 2013.

[25] [25] Y. Xie, A. Chou and D. Engler, "Archer: Using symbolic, path-sensitive analysis to detect memory access errors," ACM SIGSOFT software engineering notes, vol 28, no 5, pp. 327–336, 2003.

[26] [26] R. Xu, P. Godefroid and R. Majumdar, , "Testing for buffer overflows with length abstraction," In Proceedings of the 2008 international symposium on software testing and analysis, 2008, pp. 27–38.

[27] [27] F. Yamaguchi, N. Golde, D. Arp and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," In Security and privacy (SP), 2014 IEEE symposium on, 2014, pp. 590–604.

[28] [28] M. Zhang, Y. Duan, H. Yin and Z. Zhao, , "Semantics-aware android malware classification using weighted contextual API dependency graphs," In Proceedings of the 2014 ACM SIGSAC conference on computer and communications security, 2014, pp. 1105–1116.

[29] [29] M. Zhang and H. Yin, , "AppSealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications." In Ndss, 2014, .

[30] [30] Y. Zhang, W. Fu, X. Qian and W. Chen, "Program slicing based buffer overflow detection," Journal of software engineering and applications, vol 3, no 10, pp. 965, 2010.

[31] [31] M. Zitser, R. Lippmann and T. Leek, , "Testing static analysis tools using exploitable buffer overflows from open source code," In ACM SIGSOFT software engineering notes, 2004, pp. 97–106.

[32] [32] D. Wagner, J.S. Foster, E.A. Brewer and A. Aiken, , "A first step towards automated detection of buffer overrun vulnerabilities." In Ndss, 2000, pp. 2000–2002.

[33] [33] U. Shankar, K. Talwar, J.S. Foster and D. Wagner, "Detecting format string vulnerabilities with type qualifiers." In USENIX security symposium, 2001, pp. 201–220.

[34] [34] J.J. Tevis and J.A. Hamilton Jr, , "Static analysis of anomalies and security vulnerabilities in executable files," In Proceedings of the 44th annual southeast regional conference, 2006, pp. 560–565.