Deep Clustering-Based Vulnerability Detection Method for Open-Source Software (Postprint)

Authors: Li Yuancheng, Huang Rong, Lai Fenggang, Mao Yifan, Cai Lijun

Date: 2019-01-28T00:00:00+00:00

Abstract

Recent years have witnessed frequent disclosures of high-risk vulnerabilities in open-source software, posing significant threats to the security of enterprise information systems. To address open-source software vulnerabilities, this paper proposes a software source code vulnerability detection method based on deep clustering algorithms. This method utilizes a code graph model to construct code property graphs for open-source software, traverses to obtain key code nodes and extracts Application Programming Interface (API) sequences, embeds them into vector space, performs clustering centered on key code, calculates anomaly scores for each function based on clustering results, generates detection reports and matches them against vulnerability databases, thereby detecting vulnerabilities in source code. Experimental results demonstrate that the proposed method can locate key code segments where vulnerabilities reside in open-source software and detect corresponding vulnerabilities.

Full Text

Preamble

Open Source Software Vulnerability Detection Method Based on Deep Clustering

Li Yuancheng[1], Huang Rong[1], Lai Fenggang[2], Mao Yifan[2], Cai Lijun[3]

[1]School of Control & Computer Engineering, North China Electric Power University, Beijing 102206, China
[2]State Grid Information & Telecommunication Branch, Beijing 100761, China
[3]State Grid Fujian Information & Telecommunication Branch, Fuzhou 350003, China

Abstract: In recent years, open source software has frequently exposed high-risk vulnerabilities, posing significant threats to enterprise information system security. This paper proposes a source code vulnerability detection method for open source software based on deep clustering algorithms. The method constructs code attribute graphs from open source software using a code graph model, traverses the graphs to identify key code nodes and extracts application programming interface (API) sequences. These sequences are embedded into a vector space and clustered around key code segments. Based on the clustering results, anomaly scores are calculated for each function to generate detection reports that are matched against a vulnerability database, thereby detecting vulnerabilities in the source code. Experimental results demonstrate that the proposed method can locate vulnerable code segments in open source software and detect corresponding vulnerabilities.

Key words: open source software; vulnerability detection; source code analysis; deep learning; clustering

0 Introduction

With the rapid development of Internet and big data technologies and the continuous improvement of social informatization, open source software has become increasingly integrated into information system construction across various industries due to its characteristics of openness, sharing, commercial-free usage, and functional flexibility. Open source software refers to software that allows users to freely use and modify source code within the scope of open source licenses listed by OSI, and to combine open source code with other software code. The existence of open source software has enhanced software openness and industry development levels, promoting standardization and a virtuous cycle of software development. Due to these advantages, enterprises have continuously expanded their use of open source software in their informatization construction to save development costs and improve efficiency. Currently, open source software is used to varying degrees in critical information system components such as system frameworks, logging, data storage, processing, and transmission. Although the overall code quality of open source software continues to improve, security issues remain prevalent, exposing enterprises to numerous security risks and threats. Reports indicate that security threats from open source components and source code have grown exponentially, with high-risk vulnerabilities frequently emerging in software such as Struts2 and OpenSSL. Therefore, security detection of open source software is essential to ensure the safety and reliability of enterprise information systems.

Vulnerability research is a key focus of software security [1–4]. With the continuous development of machine learning, using machine learning algorithms to mine and analyze software vulnerabilities has attracted significant attention. Researchers have extracted code features through code metrics [5,6] and graph models [7,8], employing algorithms such as Support Vector Machines (SVM), Random Forest (RF), and nearest neighbor methods to identify vulnerabilities in open source software. Additionally, Shar et al. [9] identified vulnerabilities in open source software by extracting static code attributes based on data flow analysis as machine learning features. Scandariato and Pang et al. [10,11] introduced natural language processing techniques, such as bag-of-word and N-gram language models, to extract code features and used classification models to identify vulnerabilities in software.

Vulnerability detection using machine learning requires effective modeling, where features must include syntax, control flow, and data flow information. However, traditional metrics are not suitable for software vulnerability identification, necessitating the definition of specific metrics. Since open source software vulnerabilities typically originate from design flaws, programming errors, interaction processing defects, and logical flaws, and are associated with specific API sequences, vulnerabilities can be detected by examining API sequences of critical code segments in source code. This paper constructs source code attribute graphs through control flow and data flow analysis, extracts API sequences of critical code segments, analyzes the quantified API sequences using deep clustering algorithms [12], and calculates sequence anomaly scores to detect vulnerabilities in open source software.

1.1 Source Code Attribute Graph

Source code graph models are intermediate representations formed during program compilation, including abstract syntax trees, control flow graphs, and program dependency graphs. Abstract syntax trees display the nested structure of programs, control flow graphs indicate the execution order of statements, and program dependency graphs show the direction of data flow and control flow. Traditional code graph model generation methods require building a specific compilation environment and finding appropriate compilation tools for the programming language along with all header files. For complex or legacy programs, this process is extremely complicated and may encounter issues such as missing header files. Therefore, this paper utilizes the open-source tool Joern [13] to extract abstract syntax trees, control flow graphs, and program dependency graphs from source code and combine them into code attribute graphs. Joern is an analyzer built using Island Grammar and the parser generator tool ANTLR. Island Grammar is a syntax for describing underlying languages that enables language analysis without checking text grammar. ANTLR is an open-source parser that automatically generates syntax trees from input and provides visual representation.

In form, a code attribute graph is a labeled multigraph [14] that can store data on nodes and edges to mark relationships between nodes. For each function in the program, a code attribute graph is generated containing all nodes and relationships of that function. Each node includes primary attributes such as node type, code, and location. Each edge represents the calling sequence between nodes and the direction of data and control flow, as shown in [Figure 1: see original paper]. The red lines indicate data flow direction, blue lines indicate control flow direction, node A is the function entry, node E is the exit, and the three intermediate nodes correspond to each line of code on the right.

1.2 API Sequence Extraction

API sequences consist of API nodes, which are defined as all types and called function names contained in parameter and local variable declarations [9]. In the attribute graph, nodes at the source and convergence points of data flow and control flow are designated as "source"and "sink."In [Figure 1: see original paper], node B is the "source"and node D is the "sink."Observing the graph model reveals that the subtrees where "source" and "sink" nodes reside contain information about function parameters, variables, calls, and global and local variables, which are relevant to conditions required for secure software operations. By traversing the code attribute graph, "source"and "sink"nodes are marked, and the function node containing them is located in the abstract syntax tree. Using this node as the root, subtree nodes are extracted to form an API sequence. The subtree containing node D in the abstract syntax tree is shown in [Figure 2: see original paper].

This process is similar to program slicing, extracting code information from the subgraph where "source" and "sink" are located in the complete source code attribute graph to form an API sequence. The generated sequence includes function names of "source" and "sink," used parameters and variable information, conditional statements, and dependency relationships between nodes. Conditional statements are conditional expressions containing keywords such as if, for, and while.

1.3 Quantifying API Sequences

The API sequences obtained through the above method are abstract values composed of expressions that cannot be directly used as input for machine learning. Therefore, the sequences must be embedded into a vector space to obtain corresponding numerical vectors as sample features. Due to different coding habits among developers, function names, parameter names, and variables require normalization to obtain a unified form.

First, non-ASCII characters unrelated to vulnerability information are removed from the API sequences. Then, user-defined variables are mapped one-to-one to symbolic names (e.g., "ARG1," "ARG2"), and user-defined functions and parameters are mapped similarly (e.g., "FUN1,""PAR1"). Finally, return values are uniformly renamed (e.g., "RET1," "RET2").

After obtaining normalized sequences, they are embedded into a vector space using a mapping function. The "bag-of-word" model is introduced here, borrowing methods from natural language text processing to quantify sequences. For natural language, text consists of words, and feature vectors can be obtained by counting keyword frequencies. For API sequences, API nodes are the keywords. For each open source software, scanning the program yields the function set X and the API node set A for each source code file, which are combined to generate the API dictionary D. The mapping function ϕ can be expressed as $\phi(a) = I(x_a) \cdot w_a$, where w_a is a parameter of Term Frequency-

Inverse Document Frequency (TF-IDF) used to reduce the similarity of API sequences and eliminate the impact of common API nodes on features. Its expression is $w_a = tf_a \times \log(N/df_a)$, where tf_a is the ratio of the number of occurrences of API node a in A to the total number of nodes, and df_a is the number of files containing node a.

2.1 AE-KNN

The quantified sample features contain information required for source code defect detection. This paper also employs a KNN algorithm with neighborhood discovery to identify function neighborhoods and obtain program context information. Considering the high dimensionality of sample features in this work, an Autoencoder is introduced to compress sample features, proposing an AE-KNN deep clustering model as shown in [Figure 3: see original paper].

The model consists of three parts: sample feature preprocessing, sample feature reconstruction, and sample clustering. Preprocessing removes noise and duplicate data and transforms data into a form suitable for analysis. This paper uses Z-score normalization for sample feature preprocessing. Assuming a sample size of n and the value of the i-th feature in the sample as x_i, the normalization formula is $x_{ij} = \frac{x_{ij} - \bar{x}_i}{\sigma_i}$, where $\bar{x}_i$ and σ_i are the mean and standard deviation of the feature values, respectively.

The sample feature reconstruction part mines the preprocessed features, learns deep abstract relationships of sample features, and reconstructs low-dimensional abstract features. This paper uses the Autoencoder algorithm to reconstruct sample features. Autoencoder includes encode and decode processes, with input, output, and hidden layers. Encode compresses data, while decode reconstructs data. The sample feature reconstruction process is represented as:

$$\begin{cases} y = f_\theta(Wx + b) \\ z = g_{\theta'}(W'y + b') \end{cases}$$

where x is the input sample feature, z is the reconstructed sample feature, f_θ is the nonlinear function from input to hidden layer, y is the intermediate result, and $g_{\theta'}$ is the nonlinear function from hidden to output layer. W and W' represent encoding and decoding weights, b and b' are network bias vectors, and f and g are activation functions. This algorithm has small reconstruction error, and the compressed low-dimensional features can be considered to contain almost all feature information of the output data.

The sample clustering part clusters the obtained low-dimensional features centered on sample points where “source” and “sink” are located. Considering that vulnerability detection focuses on individual differences in code segments, cosine distance is used in experiments. Assuming x is a sample to be classified, c is a sample of a certain class, and d is the cluster center, the algorithm can be expressed as $knn(x) = \sum_{i=1}^{k} sim(x_i, d)$, where $sim(\cdot)$ is the cosine distance between x_i and d.

KNN is a lazy-learning algorithm where the classifier makes decisions based on the dominant category among k samples. This geometric representation can identify the k nearest neighbors of "source" and "sink."

2.2.1 Open Source Software Vulnerability Database

Open source software vulnerability detection methods require a vulnerability collection as a foundation, necessitating the construction of an open source software vulnerability database. Referencing well-known vulnerability databases including CVE and the National Vulnerability Database (NVD), and combining them with our methodology, we constructed an open source software vulnerability database. Currently, the database includes vulnerabilities related to input validation, buffer overflow, memory management, and API misuse. shows some key functions related to these issues.

2.2.2 Anomaly Score Calculation

Based on the clustering results of AE-KNN, anomaly scores for each function in each class are calculated. Assuming the sample set of each class is N and the vector representation of samples is x_i, considering the "source" and "sink" in each class, a normal model is constructed as $\mu = \frac{1}{N}\sum_{i=1}^{N} x_i$, where μ is the centroid vector. Each value in the vector represents neighbor scores with a numerical distribution of [0,1], indicating the percentage of functions that return check values during sample proximity checking. After obtaining the normal model, the L_∞ norm is used to calculate the anomaly score for each sample as $f(x_i) = \|x_i - \mu\|_\infty$. All samples are sorted by anomaly score, where higher rankings indicate a greater likelihood that the corresponding code segment contains vulnerabilities. High-ranking sample code fragments are extracted and matched against the open source software vulnerability database to finally obtain vulnerabilities in the tested software.

2.3 Open Source Software Vulnerability Detection Process

Based on the above description, the specific process of the proposed deep clustering-based vulnerability detection method for State Grid open source software is as follows:

a) Generate code attribute graphs from open source software source code, which contain source code data, control flow information, and code nested structures.

b) Traverse the code attribute graphs to identify "source" and "sink" nodes of data flow and control flow, extract key API sequences with these nodes as roots, embed them into vector space, and obtain feature vectors.

c) Use the AE-KNN model to perform clustering analysis on feature vectors, calculate anomaly scores for each feature sample in each obtained class, sort them, and generate detection reports.

d) Extract high-ranking sample functions from the reports, match them against the vulnerability database, and obtain vulnerabilities contained in the functions, as shown in [Figure 4: see original paper].

The AE-KNN model can determine vulnerabilities in source code through a geometric approach. Calculating cosine distances between samples in the model enables proximity discovery, quickly identifying similar API usage patterns and laying the foundation for vulnerability detection.

3.1 Experimental Data

Statistics on open source software used in State Grid Corporation' s informatization projects from July 2017 to April 2018 show that 104 open source software packages were used, accounting for 74.23% of the total. This paper selected two widely used open source software packages as experimental samples:

a) Redis: A log-type, Key-Value database written in ANSI C. Redis versions before 2.8.2 and 3.x versions before 3.0.2 lack effective authentication, allowing remote attackers to execute arbitrary Lua bytecode through the eval command (CVE-2015-4335). Version 2.8.20 was used in experiments.

b) MongoDB: A distributed file storage database written in C++. MongoDB versions before 2.4.8 and 2.6.x versions before 2.6.8 allow remote attackers to cause denial of service through special UTF-8 strings in BSON requests (CVE-2015-1609). Version 2.6.7 was used in experiments.

For these two open source software packages, function set X was constructed using the above method. By analyzing sample functions in each class where "source" and "sink" are located, function rankings were obtained. Notably, since all functions in set X are used for neighborhood selection, the rankings include both vulnerable and normal functions.

Additionally, to verify detection performance for different vulnerability types, 315 code segments were compiled from datasets provided by the National Institute of Standards and Technology (NIST) website as the experimental set, including 171 normal code segments and 144 vulnerable code segments (40 input validation, 43 buffer overflow, 33 API misuse, and 28 memory management).

3.2.1 Parameter Settings

The AE-KNN model requires determining the neighbor number k. The average clustering effect for the two open source software packages was calculated under different k values, and ROC curves were generated as shown in [Figure 5: see original paper]. The results show that the best clustering effect is achieved when $k = 25$. Therefore, $k = 25$ was selected for experiments.

3.2.2 Experimental Results Analysis

For Redis, 518 "source" and "sink" nodes were identified. Proximity discovery was performed on these 518 nodes, and anomaly scores, cosine distances, and scores were calculated and printed as shown in [Figure 6: see original paper]. Six high-ranking functions were selected from these 518 results, as shown in .

Tracing these functions revealed that, except for the fifth function in the table, all are related to vulnerability CVE-2015-4335, as shown in . In function luaV_execute, the case statement OP_FORLOOP does not perform type checking on parameters, directly calling function luai_numadd() for assignment after defining idx. Since the default parameter is lua_Number type, this leads to confusion from any type to lua_Number. Malicious users can modify instructions from other case statements to JMP instructions to skip type checking and directly enter OP_FORLOOP. Additionally, in another case statement OP_CLOSURE in function luaV_execute, a for loop for closure processing exists. Its function is to generate corresponding MOVE instructions after the CLOSURE instruction, with the second parameter of the instruction being a reference to closure variables. Under normal circumstances, this reference can only point to local variables in the current stack, but malicious users can modify bytecode to point it to any location. In function luaD_precall, the created C language closure object CClosure points to a function pointer. Malicious users can overwrite CClosur→f while obtaining the System address and overwriting it to fputs.got to exploit this vulnerability and execute arbitrary Lua bytecode through the eval command.

Similarly, MongoDB was tested, and six high-ranking functions were extracted as shown in . Except for rows 4 and 5, all these functions are related to vulnerability CVE-2015-1609. When using function validateBSON(), only length checking was performed on the original input originalBuffer before calling validateBSONIterative() to process the input data, as shown in . Additionally, in function readUTFString() called by validateBSONIterative(), after reading the string length, assignment to variable out was performed directly without judgment on sz. Due to the lack of output value checking, the MongoDB server cannot validate some malformed BSON data packets, causing authentication failures and triggering exceptions that crash the service. Attackers only need to obtain database access permissions to continuously send specially crafted data packets (specific regular expressions) to the database, causing server crashes and denial of service attacks.

For the vulnerability dataset, cross-validation was used to test the model' s accuracy. The dataset was divided into five subsets, with one subset used as the test set and the remaining as the training set each time, performing five rounds of cross-validation. Experimental results were averaged. Accuracy, False Positive

Rate (FPR), and Miss Rate were selected as evaluation metrics, calculated as:

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN}, \quad FPR = \frac{FP}{FP + TN}, \quad MissRate = \frac{FN}{TP + FN}$$

where TP is normal samples correctly detected as normal, FN is normal samples incorrectly detected as vulnerable, FP is vulnerable samples incorrectly detected as normal, and TN is vulnerable samples correctly detected as vulnerable.

Experimental results are shown in . The results demonstrate that the proposed vulnerability detection model achieves high detection accuracy and low miss rates for all four vulnerability types. However, the model exhibits higher false positive rates for buffer overflow, API misuse, and memory management vulnerabilities, possibly because key functions for some vulnerability codes overlap.

shows the AE-KNN model experimental results for different vulnerability types: Input Validation (Accuracy: 92.89%, FPR: 2.13%, Miss Rate: 11.6%), Buffer Overflow (Accuracy: 93.64%, FPR: 2.76%, Miss Rate: 13.93%), API Misuse (Accuracy: 93.03%, FPR: 2.76%, Miss Rate: 15.71%), and Memory Management (Accuracy: 93.17%, FPR: 2.88%, Miss Rate: 6.55%).

To further verify the detection performance of the proposed vulnerability detection scheme, it was compared with other vulnerability detection algorithms on the same dataset, as shown in . The results indicate that the dynamic detection scheme VDiscover performs worse than static detection schemes, while the proposed scheme achieves better detection results for multiple vulnerability types. shows the comparison results: VulDeePecker [16] (Accuracy: 87.91%, Miss Rate: 16.34%), Chucky [7] (Accuracy: 89.78%, Miss Rate: 6.55%), VDiscover [17] (Accuracy: 81.07%), and AE-KNN (Accuracy: 90.86%, Miss Rate: 6.55%).

4 Conclusion

Addressing the security issues of enterprise open source software, this paper proposes a source code vulnerability detection method based on deep clustering. The method utilizes code graph models and introduces natural text language analysis techniques to construct sample features for open source software, proposes an AE-KNN model for clustering analysis, calculates anomaly scores for each sample, sorts them to obtain critical code segments, and detects vulnerabilities in these segments by matching against a vulnerability database. The limitation of this detection model is that normal functions may exist among high-ranking functions. Future research will focus on improving model accuracy to achieve fully automated detection of vulnerabilities in State Grid open source software source code, including known vulnerabilities and zero-day vulnerabilities, and extending to open source software written in other languages.

References

[1] Cai Jun, Zou Peng, Yang Shangfei. Vulnerable spots localization methods for software vulnerability analysis [J]. Journal of National University of Defense Technology, 2015, 37(5): 141-148.

[2] Li Zhoujun, Zhang Junxian, Liao Xiangke, et al. Survey of software vulnerability detection techniques [J]. Chinese Journal of Computers, 2015, 38(4): 717-732.

[3] Xu Weiyang, Li Yao, Tang Yong, et al. Research on cross-architecture vulnerabilities searching in binary executables [J]. Netinfo Security, 2017(9): 21-25.

[4] Li Zhen, Zou Deqing, Xu Shouhuai, et al. VulDeePecker: a deep learning-based system vulnerability detection [EB/OL]. (2018-01-05). https://arxiv.org/pdf/1801.01681.pdf.

[5] Perl H, Dechand S, Smith M, et al. VCCFinder: finding potential vulnerabilities in open-source projects to assist code audits [C]//Proc of ACM SIGSAC Conference on Computer and Communications Security. New York: ACM Press, 2015: 426-437.

[6] Walden J, Stuckman J, Scandariato R. Predicting vulnerable components: software metrics vs text mining [C]//Proc of IEEE International Symposium on Software Reliability Engineering. Washington DC: IEEE Computer Society, 2014: 23-33.

[7] Wei Shengjun, He Tao, Hu Changzhen, et al. Predicting software security vulnerabilities with component dependency graphs [J]. Transactions of Beijing Institute of Technology, 2018, 38(5): 525-530.

[8] Yamaguchi F, Wressnegger C, Gascon H, et al. Chucky: exposing missing checks in source code for vulnerability discovery [C]//Proc of ACM Conference on Computer and Communications Security. New York: ACM Press, 2013: 499-510.

[9] Shar L K. Predicting common web application vulnerabilities from input validation and sanitization code patterns [C]//Proc of IEEE/ACM International Conference on Automated Software Engineering. New York: ACM Press, 2012: 310-313.

[10] Scandariato R, Walden J, Hovsepyan A, et al. Predicting vulnerable software components via text mining [J]. IEEE Transactions on Software Engineering, 2014, 40(10): 993-1006.

[11] Pang Y, Xue X, Namin A S. Predicting vulnerable software components through n-gram analysis and statistical feature selection [C]//Proc of IEEE International Conference on Machine Learning and Applications. IEEE, 2015: 543-548.

[12] Yang Qi. Research on key technologies of clustering based on deep learning [D]. Chengdu: Southwest Jiaotong University, 2016.

[13] Joern [DB/OL]. https://github.com/octopus-platform/joern.

[14] Rodriguez M A, Neubauer P. The graph traversal pattern [J]. Graph Data Management Techniques & Applications, 2010. https://arxiv.org/abs/1004.1001.

[15] Miao Xudong, Wang Yongchun, Cao Xingchen, et al. Detection approach for security vulnerability based on pattern matching [J]. Computer Science, 2017, 44(4): 109-113.

[16] Yuan Zi, Yu Lili, Liu Chao. Identification method for defect-introducing fine-grained software changes [J]. Journal of Beijing University of Aeronautics and Astronautics, 2014, 40(9): 1231-1238.

[17] Grieco G, Grinblat G L, Uzal L, et al. Toward large-scale vulnerability discovery using machine learning [C]//Proc of ACM Conference on Data and Application Security and Privacy. New York: ACM Press, 2016.

Note: Figure translations are in progress. See original paper for figures.