УДК 004

МЕТОДИ ВИЯВЛЕННЯ ПОМИЛОК БЕЗПЕКИ В ПРОГРАМНОМУ ЗАБЕЗПЕЧЕННІ НА ОСНОВІ ГЛИБИННОГО НАВЧАННЯ

А. В. Черноусов[1,2], А. Ю. Савченко[1,2], Є. Ю. Куб'юк[1,2]

[1] *Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського»*
[2] *Samsung R&D Institute Ukraine (SRK)*

Анотація

Ми представляємо VulDetect – систему виявлення вразливостей в програмному забезпеченні на основі початкового коду. Ця система використовує методи глибинного навчання для організації вирішення того, чи є фрагмент коду вразливим. Даний підхід є вдосконаленням методу, запропонованого у VulDeePecker. Модель використовує представлення початкового коду в вигляді абстрактного синтаксичного дерева. Результатом є порівняльна характеристика виявлених вразливостей обох систем за проектом Bitcoin Core.

Ключові слова: виявлення помилок, вразливості, програмне забезпечення, глибинне навчання

Вступ

На сьогодні ведеться інтенсивна розробка програмного забезпечення, які допомагають запобігати появі помилок безпеки в програмному забезпеченні. Великі компанії запроваджують процедуру Microsoft SDL [1], яка використовується для зменшення ризиків безпеки. Найбільш ефективним етапом є проведення перевірок на етапі реалізації програмного забезпечення. В даній роботі ми пропонуємо підхід для виявлення помилок безпеки в програмному забезпеченні на основі глибинного навчання, ще на етапі розробки.

Найбільш відомими методами виявлення помилок безпеки програмного коду є статичні аналізатори. Основним недоліком статичних аналізаторів є те, що для кожної помилки безпеки експерт повинен запрограмувати правило або набір правил, які зможуть знаходити помилки безпеки в програмному забезпеченні. Водночас, існують бази даних, які містять як приклад коду з дефектами, так і вже виправленого. На підставі цього, можна скористатися перевагами глибинного навчання для автоматичної побудови правил аналізу початкового коду. Основною проблемою застосування глибинного навчання для аналізу початкового коду – є представлення коду, в такому вигляді, який би дозволив узагальнити варіанти реалізації специфічних функцій з урахуванням контексту використання. Внесок цієї роботи полягає у вдосконаленні технології вилучення фрагментів коду (код-гаджетів) та представленні початкового кода для глибинного навчання. [2]

1. Пов'язані роботи

В даний час існує досить багато програмних застосунків для пошуку проблем безпеки програмного коду. Аналізатори працюють з різним поданням початкового коду: чистий початковий код, абстрактне синтаксичне дерево (АСД) та виконуваний бінарний файл [3, 4]. Зі всього списку програмних застосунків, можна виділити наступні статичні аналізатори: з відкритим початковим кодом [5, 6], комерційні продукти [7, 8] та деякі дослідницькі проекти [4, 9]. Відносно принципу роботи статичних аналізаторів можна відмітити наступні 2 підходи:

1) На основі правил [10]
2) На основі подібності коду [11].

1.1. Підхід на основі правил

Даний підхід має два основних недоліки, які полягають в інтенсивній та монотонній роботі та в високих показниках помилок другого роду.

1) Інтенсивна та монотонна робота
   В даному випадку, завдання побудови правил, за якими будуть виявлятися вразливості конкретної функції покладаються на фахівця з безпеки. Ця задаа достатньо втомлююча та суб'єктивна. Іноді виникають помилки через складність реалізованих функцій. Іншими словами, для виявлення ознак, що функція є вразливою необхідно враховувати багато аспектів. У принципі, розв'язання цієї проблеми складається з окремого написання однієї і тієї ж функції кількома експертами, а потім вибрати найбільш ефективну або застосувати комбінацію деяких функцій. Однак це призводить до ще більшої ручної праці. З цього, важливо, позбавити людей від стомлюючої і суб'єктивної задачі ручного визначення функцій, що будуть виявляти вразливості.
2) Високі показники помилок другого роду
   З іншої точки зору, рішення, що існують часто не враховують багатьох вразливостей. Іншими словами мають багато помилок другого роду.

Згідно з результатами статті [12], ці показники для Clang Analyze [6] та CppCheck [5] склали 84% та 92% відповідно. Ці значення можуть бути виправдані акцентом на низький рівень помилок першого роду, але це досі не дуже гарний показник.

1.2. Підхід на основі подібності коду

Підхід на основі подібності коду також має дві проблеми: відсутність достатнього набору даних та відсутність ефективного алгоритму.

1) Відсутність достатнього набору даних
   На даний момент не існує достатнього набору даних для такого роду досліджень. Ця проблема є актуальною, попри те, що National Vulnerability Database (NVD) [13] та NIST Software Assurance Reference (SARD) [14] стали відкритими. Так само в дослідженнях були створені бази даних, де зіставлені ідентифікатори Common Vulnerabilities та Exposures (CVE-ID) [15] з коммітами, де кожен коміт містить різницю початкового коду до виправлення та після. Але всіх цих даних все одно недостатньо для визначення усіх помилок безпеки.
2) Відсутність ефективного алгоритму
   В даний час не існує єдиного ефективного алгоритму подібності коду, який був би ефективним для всіх типів вразливостей, оскільки кожна вразливість має свої особливості, які слід брати до уваги.

Поліпшений підхід на основі подібності коду був опублікований у статті [2]. Перевагою даного методу, щодо підходу на основі подібності коду полягає у ефективному алгоритмі, заснований на глибинному навчанні. У порівнянні з підходом, що ґрунтується на правилах, він має досить низький рівень помилок другого роду, приблизно 7%. Недоліки цього методу полягають в наступному: обмежений набір даних та доволі висока кількість помилок першого роду.

Можна сказати, що системи виявлення помилок безпеки з високими показниками помилок першого роду можуть виявитися непридатними для використання, а системи з високим показанням помилок другого роду можуть виявитися марними. Це виправдовує важливість використання систем, які можуть забезпечити низький рівень помилок другого роду, поки рівень помилок першого роду не надто високий.

2. Постановка задачі

У цій роботі розглядається проблема побудови системи аналізу початкового коду, заснованого на глибинному навчанні, що дозволяє визначити, чи містить фрагмент коду помилку безпеки (витоки пам'яті, переповнення буфера, тощо). Перевагою такого підходу полягають у здатності автоматично формулювати правило для вирішення того, чи є фрагмент коду вразливим чи ні, на основі накопиченого досвіду написання коду та виправлення дефектів. Глибинні методи навчання володіють гарною здатністю вилучати узагальнені структури з великого обсягу даних, що в даному випадку є накопиченою базою кодів.

Об'єктом дослідження є процес розробки програмного забезпечення.

Предметом дослідження є виявлення помилок безпеки початкового коду програмного забезпечення.

Методи дослідження – представлення початкового коду у вигляді АСД, глибинні методи навчання для розв'язання проблем класифікації.

Наукова новизна даної роботи полягає у застосуванні АСД для перетворення початкового коду та принципу його обробки для вилучення фрагмента коду (код-гаджету), що дозволило підвищити точність виявлення дефектів. Цей підхід є вдосконаленням технології, описаної раніше в статті [2].

3. Загальна архітектура

Нашою метою є розробка системи виявлення вразливостей (VulDetect), яка може автоматично визначити, чи є дана програма в вигляді початкового C/C++ коду вразливою, і якщо так, то місця розташування вразливостей. Цього слід досягти, не вимагаючи від людини визначати функції вручну і без високих показників помилок другого роду (до тих пір, поки рівень помилок першого роду не достатньо високий). У цьому розділі ми описуємо загальну архітектуру VulDetect. Почнемо з обговорення поняття гаджета коду, оскільки воно є вирішальним для представлення програм.

3.1. Вилучення код-гаджетів

Працюючи з початковим кодом, ми повинні вирішити, з якою частиною коду ми будемо працювати, це може бути функції, деякі рядки або ж увесь файл. Нам потрібно вилучити з початкового коду деякий фрагмент, який буде пов'язаний з потенційною вразливістю та з яким ми будемо працювати надалі. Цей фрагмент коду буде назватися код-гаджет за аналогією зі статтею [2]. Код-гаджет – це набір операторів, пов'язані з потоком даних. Вилучати код-гаджет ми будемо не з початкового коду, а з заздалегідь перетвореним в АСД. АСД – це структура даних, яка представляє початковий код у вигляді дерева, де кожен вузол асоціюється з мовною конструкцією, яка трапляється в початковому коді. Алгоритм вилучення код-гаджетів демонструється на Рисунку 1:

1) Попередня обробка. На основі початкового коду, використовуючи вбудовану попередню обробку clang, створюється новий файл без директив препроцесора, які були визначені користувачем.
2) Побудова АСД. Для побудови АСД ми використовуємо clang compiler toolkit.
3) Пошук початкових точок. Під початковою точкою розуміється місце в початковому коді, з якого починається аналіз. Ми використовуємо виклики функцій зі стандартної бібліотеки C/C++ як початкову точку (наприклад, malloc, memcpy, fopen і т.д.)

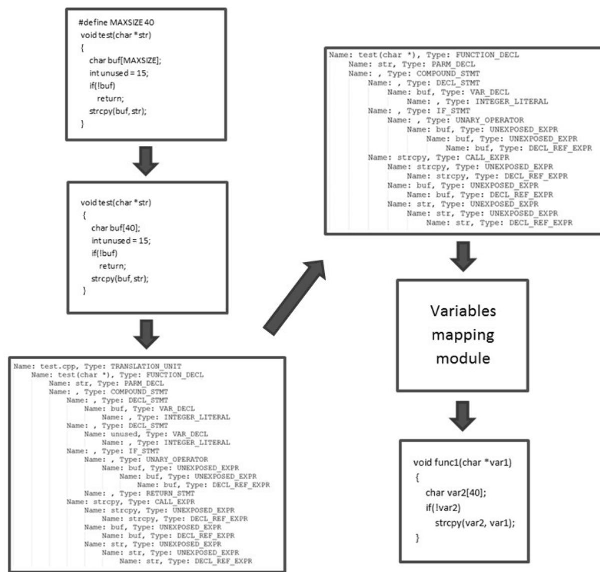

Рис. 1. Кроки для вилучення код-гаджетів

4) Створення графу залежностей. На цьому етапі ми будуємо граф залежності аргументів (або повернених значень) від початкової точки.
5) Перейменування змінних або функцій. На цьому кроці ми позбуваємося залежностей від користувацьких імен функцій і змінних. Ми замінюємо всі імена функцій і змінних символічними іменами, такими як: «var1», «var2», і т.д та «func1», «func2», і т.д. Водночас, це призводить до зіставлення усіх код-гаджетів в один вигляд.
6) Створення код-гаджету. На основі графу залежностей, код-гаджет формується з токенів, взятих з вузла графу.

3.2. Деталі підходу

Після вилучення код-гаджетів з початкового коду, ми перетворюємо його у числовий вектор, індекси якого будуть відповідати номеру рядка коду в поточному код-гаджеті. Щоб досягти бажаного результату, ми використали word2vec [16], що дозволило зберегти лексичні та логічні зв'язки в межах вектору для кожного сегмента коду. Далі, ми використали модель нейронної мережі з вчителем, де кожен вхідний вектор був позначений як 1 або 0, відповідно, вразливий, або ні. Була обрана наступна структура моделі, відношення кількості епох навчання до кількості навчальних прикладів, дорівнювало 100 та складалась з частин що зображені на Рисунку 2:

1) п'ять BLSTM шарів, на вході і виході якого вектор розміру N
2) один dense layer типу «rele»
3) один dense layer типу «softmax», який приймає на вхід вектор розмірності N_features та скорочує їх до 6-ти
4) останній прошарок використовував функцію «binary_crossentropy» та результатом є число типу «float» в діапазоні від 0 до 1.

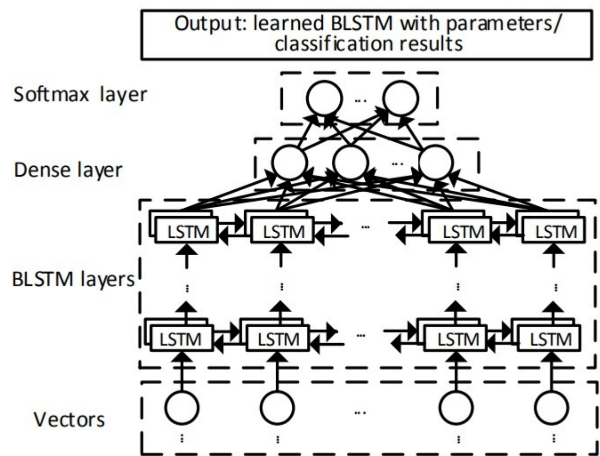

Рис. 2. Схема нейронної мережі BLSTM

4. Результати дослідження

У ході дослідження було зроблено порівняння двох моделей: VulDeePecker і VulDetect. Для підготовки моделі VulDeePecker в якості навчального набору даних було обрано попередньо підготовлені код-гаджети від авторів статті [2]. Для навчання моделі VulDetect були створені власні код-гаджети з використанням вищезазначених алгоритмів. Набір даних було створено на основі початкових кодів, взятих з National Vulnerability Database (NVD) [13], а також з NIST Software Assurance Reference (SARD) [14]. Набір даних містить 8122 зразки коду з наявністю вразливості переповнення буфера, а також 1729 зразків коду з вразливостями, пов'язаними з неправильним управлінням ресурсами. Набір даних поділили для тренування і тестування у співвідношенні 80% і 20% відповідно. Порівняльні характеристики точності навчання і тестування для моделей VulDeePecker i VulDetect наведено в Таблиці 1.

Табл. 1. Порівняння точності моделей на різних етапах

Етап/точність	**VulDeePecker**	**VulDetect**
Навчання	96%	97%
Тестування	91%	92%

Як видно з Таблиці 1, точність моделі VulDetect перевищує точність моделі VulDeePecker. Це пов'язано з використанням іншого запропонованого алгоритму для вилучення код-гаджета.

У якості експерименту, навчені моделі VulDeePecker i VulDetect використовувалися для виявлення помилок безпеки у початкових кодах реалізації Bitcoin Core [17]. Результати представлені в Таблиці 2.

За результатами експерименту, зрозуміло, що модель VulDeePecker позначила як «вразливе» набагато більшу кількість код-гаджетів, ніж модель VulDetect, але з більш низьким індексом аномалій, що вказує на більшу кількість помилок першого роду.

Табл. 2. Порівняння VulDeePecker та VulDetect за різними показниками

Показники	VulDeePecker	VulDetect
Кількість унікальних код-гаджетів	1510	
Кількість код-гаджетів відмічені як «безпечні»	970	1326
Кількість код-гаджетів відмічені як «вразливі»	540	184
Кількість код-гаджетів відмічені як «безпечні» обома моделями	802	802
Кількість код-гаджетів відмічені як «вразливі» обома моделями	16	16
Кількість унікальних код-гаджетів відмічені як «безпечні»	168	524
Кількість унікальних код-гаджетів відмічені як «вразливі»	524	168

Висновки

У даній роботі представлена нова система виявлення помилок безпеки у початковому коді на основі глибинного навчання, що називається VulDetect. Ця система є вдосконаленням технології VulDeePecker. Модель використовує представлення початкового коду у вигляді АСД. У ході дослідження було розроблено нову систему вилучення код-гаджетів. Було порівняно обидві системи за результатами виявлення помилок у початковому коді на проекті Bitcoin Core[17]. У моделі VulDetect знизилися показники помилок першого роду, що призвело до збільшення помилок другого роду. Досі існує проблема з правильним набором даних, вирішення якої дозволить підвищити точність прогнозування нейронних мереж і збільшити кількість виявлених вразливостей. У майбутньому планується зробити більш ефективним алгоритм вилучення код-гаджетів, що дозволить підвищити точність виявлення вразливостей у початковому коді.

Перелік використаних джерел

1. Microsoft security development lifecycle (SDL). — Access mode: http://www.cs.fsu.edu/~jowett/MS_SDL_Version_3.2.pdf.
2. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection / Zhen Li, Deqing Zou, Shouhuai Xu et al. // Proceedings 2018 Network and Distributed System Security Symposium. — Internet Society, 2018. — Access mode: https://doi.org/10.14722/ndss.2018.23158.
3. Neural Nets Can Learn Function Type Signatures From Binaries / Zheng Leong Chua, Shiqi Shen, Prateek Saxena, Zhenkai Liang // 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017. / Ed. by Engin Kirda, Thomas Ristenpart. — USENIX Association, 2017. — P. 99–116. — Access mode: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/chua.
4. Yamaguchi Fabian, Lottmann Markus, Rieck Konrad. Generalized vulnerability extrapolation using abstract syntax trees // Proceedings of the 28th Annual Computer Security Applications Conference on - ACSAC '12. — ACM Press, 2012. — Access mode: https://doi.org/10.1145/2420950.2421003.
5. Marjamaki D. Cppcheck - a tool for static c/c++ code analysis. — Access mode: http://cppcheck.wiki.sourceforge.net/.
6. Fandrey B.Sc. Dominic. Clang/LLVM. — Access mode: https://llvm.org/pubs/2010-06-06-Clang-LLVM.pdf.
7. Coverity. — Access mode: https://scan.coverity.com/.
8. Dexter. — Access mode: https://github.com/Samsung/Dexter.
9. VulPecker / Zhen Li, Deqing Zou, Shouhuai Xu et al. // Proceedings of the 32nd Annual Conference on Computer Security Applications - ACSAC '16. — ACM Press, 2016. — Access mode: https://doi.org/10.1145/2991079.2991102.
10. Static Code Analysis. — Access mode: https://www.owasp.org/index.php/Static_Code_Analysis.
11. Comparison and Evaluation of Clone Detection Tools / Stefan Bellon, Rainer Koschke, Giulio Antoniol et al. // IEEE Transactions on Software Engineering. — 2007. — sep. — Vol. 33, no. 9. — P. 577–591. — Access mode: https://doi.org/10.1109/tse.2007.70725.
12. Athos Ribeiro Paulo Meirelles Nelson Lago, Kon Fabio. Ranking source code static analysis warnings for continuous monitoring of FLOSS repositories. — Access mode: https://www.oss2018.org/wp-content/uploads/2018/06/Athos-Ribeiro-oss.pdf.
13. NATIONAL VULNERABILITY DATABASE. — Access mode: https://nvd.nist.gov/.
14. NIST Software Assurance Reference Dataset. — Access mode: https://samate.nist.gov/SARD/.
15. Common Vulnerabilities and Exposures. — Access mode: https://cve.mitre.org/.
16. word2vec. — Access mode: https://code.google.com/archive/p/word2vec/.
17. Bitcoin Core. — Access mode: https://github.com/bitcoin/bitcoin.