

СИСТЕМА АНАЛІЗУ БІНАРНИХ ВРАЗЛИВОСТЕЙ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

В. В. Карко¹, М. В. Грайворонський¹

¹Національний технічний університет України «Київський політехнічний інститут»

Анотація

В роботі пропонується автоматизована система аналізу бінарних вразливостей програмного забезпечення для Linux-подібних систем, основним завданням якої є виявлення та ідентифікація помилок наявних на рівні машинного коду родини процесорів Intel x86, які можуть бути використанні зловмисником для компрометації безпеки. Пропонується застосування графу залежності, який дозволяє значно звузити область пошуку вразливостей, прискорити та полегшити роботу фахівця з безпеки.

Ключові слова: аналіз бінарних вразливостей, статичний аналіз

Вступ

Створення програмного забезпечення неодмінно супроводжується внесенням помилок. Значна кількість помилок, за певних умов, може призвести до збоїв в роботі програми та становити загрозу безпеці інформації. На етапі тестування виявляється та усувається лише частина помилок, інша потрапляє до готового продукту.

Сучасні засоби розробки успішно виправляють синтаксичні помилки та проводять статичний аналіз вихідного коду високого рівня (C/C++, Java, Python, тощо) на наявність потенційних логічних помилок [1, 2]. Але через різноманіття технологій, платформ та архітектур, вихідний код проходячи безліч трансформацій та оптимізацію, зрештою компілюється в машинний бінарний код. Виникає невідповідність між тим, що програміст мав намір реалізувати і що насправді виконується процесором, тому будь-який аналіз помилок на рівні програмного коду не є вичерпним [3].

Існують принципово відмінні підходи до аналізу вразливостей в програмному забезпеченні – *статичний* та *динамічний* аналіз.

Статичний аналіз проводиться лише на основі вхідного або машинного коду програми, без безпосереднього запуску програми, а тому характеризується високим рівнем похибок першого роду та низьким рівнем похибок другого роду. Тобто, статичний аналіз, найвірогідніше, виявить вразливість наявну в програмі, проте частина виявлених «вразливостей» будуть відповідати коректному коду.

Динамічний аналіз проводиться шляхом спостереження за запущеною програмою. Характерним є аналіз лише поточної гілки програми для певної умови, що збільшує ймовірність пропустити наявну вразливість в інших гілках (високий рівень похибок другого роду), та зменшує ймовірність хибно прийняти за вразливість коректний код (низький рівень похибок першого роду).

В роботі пропонується автоматизована система статичного аналізу бінарних вразливостей програмного забезпечення для Linux-подібних систем, основним завданням якої є *виявлення та ідентифікація* наявних помилок на рівні машинного коду родини процесорів Intel x86 які можуть бути використанні зловмисником для компрометації безпеки.

Система націлена на *виявлення* потенційно небезпечних недоліків з метою їх усунення, на відміну від інших робіт із статичного аналізу [4, 5], які покладаються на *символічний аналіз*. В даній роботі пропонується застосування *графу залежності*, який виявляє потенційні вразливості, не зважаючи на наявність в програмі умовних циклів, непрямих викликів та переходів, і тим самим забезпечує краще покриття ніж символічний аналіз.

1. Етап дизасемблювання

На першому етапі відбувається зчитування заголовку бінарного файлу і визначаються: точка входу в програму, сегменти даних та коду, таблиці зв'язаних процедур та імпортованих функцій. Потім відбувається дизасемблювання (трансляція) машинного коду у відповідний асемблерний код [6].

Адреси викликаних функцій співставляються із даними в таблиці імпортованих функцій, з якої визначається назва функції.

Комплексні асемблерні інструкції розкладаються на множину простіших інструкцій. Наприклад, інструкції роботи із стеком PUSH та POP.

Таким чином, отримується множина асемблерних інструкцій $I = \{i_1, \dots, i_n\}$ для даної програми P .

В роботі застосовується лінійний алгоритм дизасемблювання, оскільки таким чином покривається весь код програми. На подальших етапах відбувається розпізнавання потоку програми та даних.

```

▲: CALL:0x8048b9d
function6:
0x80487d3:    PUSH    ESP1730, EBP306
0x80487d4:    MOV     EBP307, ESP308
0x80487d6:    SUB     ESP309, 0x18
0x80487d9:    MOV     DWORD PTR SS:[ESP313]311, _in_char* format = 0x8048f4c "\tName: "
0x80487e0:    CALL    0x80484f0 <printf@plt>
▼: NONE:0x80487e5

▲: NONE:0x80487e9
0x80487e5:    MOV     EAX317, DWORD PTR DS:[0x804b060]318
0x80487ea:    MOV     EDX325, function6.arg0 = DWORD PTR SS:[EBP323 + 0x8]321
0x80487ed:    ADD     EDX326, 0x8
0x80487f0:    MOV     DWORD PTR SS:[ESP330 + 0x8]328, _in_FILE* stream = EAX332
0x80487f4:    MOV     DWORD PTR SS:[ESP335 + 0x4]333, _in_int num = 0x40
0x80487fc:    MOV     DWORD PTR SS:[ESP340]338, _out_char str = EDX342
0x80487ff:    CALL    0x8048510 <fgets@plt>
[!]: untrusted data consumed at 0x8048c22;
▼: NONE:0x8048804

```

Рис. 1. Результат аналізу декількох базових блоків програми із CSAW CTF 2015, де виявлено постачальника даних

```

▲: NONE:0x8048c0b
0x8048c10:    MOV     DWORD PTR SS:[ESP1225]1223, _in_ ... = 0x8049099 "\tDescription: "
0x8048c17:    CALL    0x80484f0 <printf@plt>
▼: NONE:0x8048c1c

▲: NONE:0x8048c17
0x8048c1c:    MOV     EAX1233, function9.arg3 = DWORD PTR SS:[EBP1231 + 0x14]1229
0x8048c1f:    MOV     DWORD PTR SS:[ESP1236]1234, _in_char* format = EAX1238
0x8048c22:    CALL    0x80484f0 <printf@plt>
[!]: untrusted data at 0x80487ff supplied to printf;
[!]: untrusted data at 0x80488bb supplied to printf;
[!]: untrusted data at 0x8048946 supplied to printf;
[!]: untrusted data at 0x8048a68 supplied to printf;
[!]: untrusted data at 0x8048b17 supplied to printf;
▼: NONE:0x8048c27

```

Рис. 2. Результату аналізу декількох базових блоків програми із CSAW CTF 2015, де виявлено споживача даних

2. Етап побудови потоку управління

Побудова потоку управління є необхідним етапом для виявлення вразливостей, оскільки дозволяє розбити програму на *базові блоки*, які не містять переходів, що виявляє зв'язки ділянок між собою та дозволяє проводити подальший аналіз в їх межах.

Базовий блок B_k – це послідовність інструкцій програми:

$$B_k = [i_k, \dots, i_{k+n-1}, i_{k+n}] \subseteq I, n > 0$$

в якій: $[i_k, \dots, i_{k+n-1}] \not\subseteq T \wedge i_{k+n} \in T$, де $T = \{\text{CALL}, \text{JMP}, \text{Jcc}\}$ – команди що впливають на зміну послідовності інструкцій: виклик процедури, умовні та безумовні переходи.

Потік управління $C(P) = \langle B, E \rangle$ програми P – орієнтований граф із множини вершин базових блоків B та множини ребер $E = \{(b_i, b_j)\}$ – перехід управління від блоку b_i до b_j .

Більшість інструкцій здійснюють перехід на задану адресу, або на наступну інструкцію, таким чином

базові блоки переважно мають одне або два вхідних та вихідних ребер.

Для виявлення користувацьких функцій застосовується алгоритм пошуку *компонент зв'язності* із застосуванням проходження графу C в глибину, при цьому ребра від інструкції CALL не враховуються.

Основною проблемою для побудови потоку управління є *непрямі переходи* та *непрямі виклики* – це такі інструкції, які можуть перейти на адресу, яка вираховується під час роботи програми. Тому в загальному випадку, визначення адреси непрямих переходів або викликів статичним методом неможливе. Часткове рішення засноване на тому факті, що блоки, на які можливий перехід, хоч і не мають вхідних ребер від блоку із непрямим переходом, проте мають вихідний спільний блок. Тобто, такі базові блоки зв'язані деякими спільними даними, а тому розміщуються в межах однієї функції.

3. Етап побудови потоку даних

Побудова потоку даних є одним із найскладніших та важливіших етапів. На цьому етапі формуються зв'язки між тим як дані пов'язані між собою в межах базових блоків, в межах функцій та зв'язки між функціями.

Відповідно до семантики кожної інструкції визначаються визначені та використані операнди.

Операнд (регістр процесора або комірка пам'яті) називається *визначеним* (англ. *defined*), якщо інструкція змінює його вміст, або *використаним* (англ. *used*), якщо зчитує.

Потік даних $D(P) = \langle O, E \rangle$ програми P – орієнтований граф із множини вершин операндів O та множини ребер $E = \{(o_i, o_j) - \text{зв'язок між операндом } o_i \text{ та } o_j\}$.

Першим кроком, будуються ребра між операндами в межах однієї інструкції, відповідно до її семантики.

Другим кроком, будуються ребра між операндами в межах базового блоку: спочатку між регістрами, потім – між комірками пам'яті.

В архітектурі Intel x86 комірка пам'яті може адресуватися базовим регістром *base*, індексним регістром *index* з множителем *scale* $\in \{0, 1, 2, 4, 8\}$ та зміщенням *offset* і в загальному випадку має вигляд:

$$memory = [base + index \times scale \pm offset]$$

Виходячи з цього, вважаємо, що дві комірки пам'яті m_1 та m_2 звертаються до одної і тої самої фізичної комірки, якщо:

$$m_1 \equiv m_2 \Leftrightarrow \begin{cases} \exists base_1 \rightarrow base_2 \\ \exists index_1 \rightarrow index_2 \\ offset_1 = offset_2 \end{cases}$$

де: $x \rightarrow y$ – маршрут від вершини x до y в графі D .

Останнім кроком, будуються ребра між аргументами функції та поверненим значенням функції, відповідно до визначеної конвенції про виклики функції (*cdecl* та *stdcall*), та визначеним сигнатурам стандартних функцій із бази сигнатур.

На рис. 1 зображено уривок результату аналізу декількох базових блоків ($B_{0x80487a3}$, $B_{0x80487e5}$) однієї із ідентифікованих користувацьких функцій *function6(arg0)* програми із CSAW CTF 2015.

Результатом аналізу потоку даних блоку $B_{0x80487e5}$ є підграф потоку даних на рис. 3:

4. Етап виявлення вразливостей

На даному етапі виконується пошук вразливостей застосовуючи отримані на попередніх етапах графі потоку управління та потоку даних.

Для різних класів вразливостей можуть використовуватися різні моделі аналізу. В роботі приведена модель для пошуку вразливостей, заснованих на потраплянні неперевіраних користувацьких даних до чутливих функцій. Типовим представником цього класу є вразливість формату стрічки (англ. *format string vulnerability*) [7].

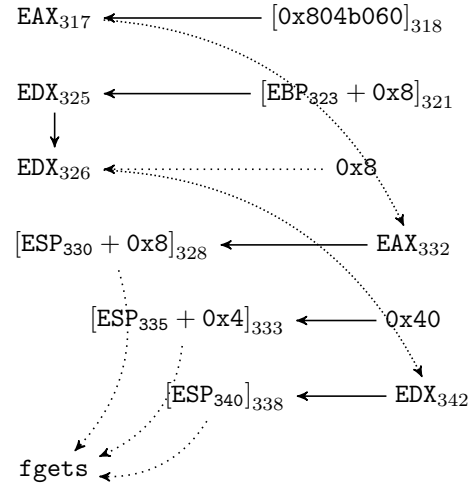


Рис. 3. Приклад підграфу потоку даних

Першим кроком, визначаються підмножини *постачальників* S і *споживачів* C користувацьких даних із множини вершин графу потоку даних D .

Постачальниками користувацьких даних є: аргументи командної стрічки, змінні оточення, буфери в функціях зчитування із потоку введення, файлу, сокету (*gets()*, *fgets()*, *read()*), тощо. Прикладом (див. рис 1) є операнд EDX_{342} , який є стрічковим буфером, в який заноситься інформація від користувача після виклику функції *fgets()* за адресою $0x80487ff$.

Споживачами є: чутливі функції із виведення тексту в потік, файл, буфер (*printf()*, *fprintf()*, *sprintf()*), функція виклику системних команд *system()*, та виклику процесу *open()*, тощо. Прикладом (див. рис 2) є операнд EAX_{1238} , в якому міститься інформація від користувача і потрапляє до функції *printf()* за адресою $0x8048c22$.

Другим кроком, перевіряється наявність маршруту від кожного постачальника до кожного споживача із застосуванням проходження графу D в ширину. Наявність маршруту свідчить про потрапляння неперевіраних даних користувача до чутливих функцій, що може становити вразливість. Будемо предикат $V(s, c)$, $s \in S$, $c \in C$ для визначення потенційної вразливості:

$$V(s, c) = \begin{cases} 1 & \text{якщо } \exists s \rightarrow c \\ 0 & \text{інакше} \end{cases}$$

Тоді потенційні вразливості цього класу:

$$V_{untrust}(P) = \{(s, c) \mid \forall s \in S : \forall c \in C : V(s, c) = 1\}$$

Загальна множина вразливостей $V(P)$ програми P :

$$V(P) = \bigcup V_i(P)$$

де $V_i(P)$ – потенційні вразливості виявленні моделлю аналізу i .

5. Оцінка

Для проведення оцінювання було створено демонстраційну модель із реалізацією зазначених етапів. Тестування проводилося на трьох програмах: програми із конкурсу з кібербезпеки LSE CTF 2014

[8], CSAW CTF 2015 [9] та програма із комплекту GNU Sharutils 4.2.1 [10]. Кожна з цих програм має по одній відомій вразливості типу формату стрічки. Дані результату аналізу наведено в таблиці 1.

Програма P	Інструкцій	$ S $	$ C $	$ V(P) $
LSE	357	8	4	1
CSAW	618	16	7	5
Sharutils	5056	78	3	6

Табл. 1. Результату аналізу

Порівнюючи з методом повного перебору M_{force} (який позначить $|S| \cdot |C|$ вразливостей) та пропонованим методом M (див. табл. 2), можна стверджувати, що пропонований метод дозволяє значно звузити область пошуку вразливостей.

Програма P	M_{force}	M	α_{force}	α
LSE	32	1	3.12 %	100 %
CSAW	112	5	0.89 %	20 %
Sharutils	234	6	0.43 %	16 %

Табл. 2. Порівняння запропонованого методу з методом повного перебору

Висновки

В роботі запропонована автоматизована система аналізу бінарних вразливостей програмного забезпечення для виявлення та ідентифікації потенційних вразливостей на рівні машинного коду. В основі покладено застосування *графу залежності*, що забезпечує краще покриття ніж символічний аналіз.

Було реалізовано демонстраційну модель і проведено тестування на декількох реальних програмах. Отримані результати показують, що метод має похибки першого роду, адже не всі ідентифіковані вразливості насправді є вразливістю. Однак, можна стверджувати, що пропонований метод дозволяє зна-

чно звузити область пошуку вразливостей, що прискорить та полегшить роботу фахівця з безпеки.

Перелік використаних джерел

1. Veracode. — <http://www.veracode.com/>.
2. IDA Pro. — <https://www.hex-rays.com/products/ida/>.
3. Balakrishnan Gogul, Reps Thomas. WYSINWYX: What You See is Not What You eXecute // ACM Trans. Program. Lang. Syst. — 2010. — Aug. — Vol. 32, no. 6. — P. 23:1–23:84.
4. Static Detection of Vulnerabilities in x86 Executables / Marco Cova, Viktoria Felmetzger, Greg Banks, Giovanni Vigna // Computer Security Applications Conference / Department of Computer Science. — University of California, Santa Barbara, USA : IEEE, 2011. — Dec. — P. 269–278.
5. AEG: Automatic Exploit Generation / Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, David Brumley // Network and Distributed System Security Symposium. — Carnegie Mellon University, Santa Barbara, USA, 2011. — Feb.
6. Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual. No. 325462-055US. — 2015. — June.
7. Erickson Jon. Hacking: The Art of Exploitation, 2Nd Edition. — Second edition. — San Francisco, CA, USA : No Starch Press, 2008. — ISBN: 9781593271442.
8. LSE CTF. — <https://ctf.lse.epita.fr/ex/5/>. — 2014.
9. CSAW CTF. — <https://github.com/ctfs/write-ups-2015/tree/master/csaw-ctf-2015/pwn/contacts-250>. — 2015.
10. Offensive Security Exploit Database Archive. — <https://www.exploit-db.com/exploits/479/>.