УДК 004.056:004.415.532(045)

Ануфрієнко К.П., Бобровський В.В., Луценко Д.В., Григоренко О.В.

КАТЕГОРИЗАЦІЯ МЕТОДИК ФАЗЗІНГУ

Вступ. Статичний аналіз початкового коду як метод виявлення урзливостей — це метод білого ящика. Перевірка початкового коду при цьому вимагає того, щоб початковий код був доступний. Проте існують альтернативні методи чорного ящика, при яких доступ до початкового коду не потрібен. Одна з таких альтернатив — технологія фаззінгу, яка чудово себе зарекомендувала при знаходженні серйозних уразливостей, які іншими методами не вдалося виявити [1].

Фаззінг (fuzzing) — це метод виявлення уразливостей у програмному забезпеченні за допомогою передачі йому різноманітних несподіваних вхідних даних та моніторингу винятків, тобто проблемних реакції під час виконання. Звичайно це автоматизований або напівавтоматизований процес. Це досить загальне визначення, але воно висвітлює основну концепцію фаззінгу [1].

Фаззінг був вперше запропонований в дослідницькій роботі 1990 року під керівництвом професора Бартона Міллера [2]. Протягом 90-х років XX ст. фаззінг розвивався досить мляво. Дослідження в сфері фаззінгу пожвавішали протягом першого десятиліття XXI ст. і продовжують набирати оберти. Незважаючи на такий значний розвиток, фаззінг як технологія ще тільки розвивається. Більшість розроблених до цього часу засобів — це досить невеликі проекти, створені дрібними групами дослідників або взагалі одним програмістом [1]. Тільки в останні декілька років дослідження у сфері фаззінгу знайшли комерційне застосування.

Отже, проведено чимало різноманітних досліджень щодо застосування фаззінгу у виявленні уразливостей в програмах, і виникає актуальна потреба впорядкувати результати цих робіт з метою покращення ефективності нових розроблюваних засобів фаззінгу, що в свою чергу сприятиме підвищенню захищеності програмного забезпечення. Тому **завданням** статті є категоризація відомих методик фаззінгу та з'ясування суті специфічних ключових понять, які розкривають концепцію та зміст цих категорій методик.

Проаналізувавши низку джерел [1–35], можна загалом усі дослідження у сфері фаззінгу структурувати на такі узагальнюючі категорії методик:

1) традиційний фаззінг;

2) фаззінг-методики білого ящика, засновані на символічному виконанні;

3) методики безпосереднього фаззінгу, засновані на аналізі вад;

4) методики аналізу протоколів та форматів вхідних даних для керівництва фаззінгом.

Традиційний фаззінг. Перший простий засіб фаззінгу, представлений у роботі [2], тільки породжував потоки випадкових символів і надсилав їх у цільові програми, але цей інструмент зміг викликати аварійне завершення 25-33% утиліт в системі UNIX. З тих пір був розроблений широкий спектр засобів фаззінгу.

Щоб здійснювати фаззінг, потрібні специфічні вхідні дані. Всі методики фаззінгу ґрунтуються на одному з двох наступних підходів до підготовлення вхідних даних, які були встановлені у працях [1] та [3]:

1) мутування вхідних даних — полягає у модифікації існуючих зразків даних;

2) породження вхідних даних — полягає у створенні умов для тестування з чистого аркушу за допомогою моделювання даних (протоколу або формату файлу).

Так, формувач (наприклад, Spike [1; 2], Peach [1], SNOOZE [5]) створює неправильний вхідні дані на основі визначених специфікації і пізніше (наприклад, FileFuzz [1]), мутує добре сформовані вхідні дані.

Тим не менше, загальний недолік традиційних методів фаззінгу полягає у тому, що обробка програмою більшості неправильних вхідних даних передчасно переривається. Для підвищення ефективності інструментів фаззінгу наступні категорії нових методик були запропоновані.

Фаззінг-методики білого ящика, засновані на символічному виконанні (Symbolic-execution-based white-box fuzzing). Цей метод широко реалізований в низці програмних засобах, таких як CUTE [6], DART [7], EXE [8], SAGE [9], KLEE [10] та SmartFuzz [11]. У цілому, засновані на оснащенні засобами контролю коду або відстеженні програм, ці засоби заміняють конкретні вхідні дані символічними значеннями, збирають і вирішують обмеження на виконання програмних слідів і керують виявленням та породженням помилкових вхідних даних. Для випадку, коли тестуються застосунки з високо структурованими вхідними даними, такі як компілятори та інтерпретатори, у роботах [12] та [13] запропоновано варіант методу, який використовує вхідні символічні специфікації граматики. Ці засоби доведено дуже підвищили ефективність традиційних засобів фаззінгу. Вони успішно виявили серйозні помилки в основних утилітах GNU [10], Windows-застосунках [9; 12], і файлових системах Linux [8].

З нульовим знанням про алгоритм контрольної суми, автоматичне породження тестових випадків з правильними полями контрольних сум є великою проблемою. Тестовий випадок (test case) у розробці програмного забезпечення — це набір умов, за яких тестувальник буде визначати чи задовольняється заздалегідь певна вимога. Щоб визначити, що вимога повністю виконується, може знадобитися багато варіантів тестування [14].

Для простих алгоритмів контрольної суми існуючі системи символічного виконання, такі як Replayer [15], можуть автоматично генерувати правильні контрольні суми. Однак, як показано в [15], обчислення контрольної суми значно збільшує складність зібраної символічної формули. Крім того, попередні дослідження [16; 17] показали, що нинішні двигунці символічного виконання і вирішувачі обмежень не можуть точно генерувати і вирішувати обмеження, які описують повний процес складних алгоритмів контрольної суми. У роботі [15] також запропоновано прийнятну схему для вирішення проблеми складних алгоритмів контрольної суми, тобто ітеративне обмеження деяких вхідних змінних, щоб мати конкретні значення, а потім спрощення і рішення обмежень слідів. Замість цього, TaintScope безпосередньо залишає всі байти конкретними, за винятком полів контрольних сум, що значно знижує складність обмежень слідів.

Методики безпосереднього фаззінгу, засновані на аналізі вад (Taint-analysis-based directed fuzzing). Перевірка на вади (taint checking) — засіб у деяких комп'ютерних мовах програмування, таких як Perl та Ruby, призначений для підвищення безпеки, перешкоджаючий зловмисникам виконати команди на комп'ютері. Перевірка на вади виділяє конкретні ризики для безпеки в першу чергу, пов'язані з веб-сайтами, які піддаються нападу з використанням методів, таких як SQL-ін'єкція або переповнення буферу.

Основна ідея перевірки на вади полягає у тому, що будь-яка змінна, яка може бути модифікована зовнішнім користувачем (наприклад, змінна, яка задається у полі веб-форми) являє собою потенційний ризик для безпеки. Якщо ця змінна використовується у виразі, який визначає іншу змінну, ця інша змінна тепер також підозріла. Засіб перевірки на вади проходить змінну за змінною до тих пір поки не завершить складання переліку усіх змінних, які потенційно знаходяться під впливом зовнішнього введення. Якщо будь-яка з цих змінних використовується для виконання небезпечних команд (таких як безпосередня команда до бази даних SQL або операційної системи комп’ютера), засіб перевірки на вади попереджає програму, що вона використовує потенційно небезпечну «зіпсовану» змінну. Програміст має переконструювати програму, щоб створити надійну стіну навколо небезпечного введення [18].

Перевірка на вади може розглядатися як традиційна апроксимація повної верифікації невтручання або більш загальної концепції захищеного потоку інформації [19]. Оскільки потік інформації у системі не може бути перевірений шляхом контролю одинарного трасування виконання цієї системи, результат аналізу вад неминуче відображатиме наближену інформацію відносно характеристик інформаційного потоку системи, до якої він застосований [20].

Таким чином, перевірка на вади — це метод складання чорних списків змінних, які небезпечні.

BuzzFuzz [21] — це засіб фаззінгу методом білого ящика, заснований на безпосередньому динамічному аналізі вад. Щоб відстежувати приховану інформацію, BuzzFuzz необхідно

оснащення початковим кодом застосунка. Оснащені програми відповідають за визначення вхідних даних, які можуть вплинути на значення системних викликів. Проте, сучасні програми широко використовують сторонні бібліотеки. BuzzFuzz не може бути оснащеним такими бібліотеками, якщо їх початковий код недоступний, що призводить до втрати інформації про вади. На відміну від BuzzFuzz, TaintScope безпосередньо опрацьовує бінарний виконуваний код і може спостерігати виконання всіх бібліотек. Крім того, TaintScope може обійти перевірку контрольної суми в програмах [22].

Flayer [23] — це засіб відстеження вад з можливістю перенаправлення потоків виконання під час зміни команд умовного переходу. Flayer заснований на функціональності Memcheck [24], і помічає вхідні дані лише міткою 0/1. Таким чином, Flayer не може коректно відстежувати вплив вхідних даних на виконання програми. Перетворення потоку виконання також використовується в аналізі шкідливої поведінки, наприклад, вивчення кількох шляхів виконання [25], і примусова подача пробного виконання для виявлення різних поведінок руткіта ядра [26].

Corpus Distillation [27] — це система фаззінгу зі зворотнім зв'язком, яка використовує евристичне покриття коду для вибору і мутування вхідних зразків.

Під покриттям коду (code coverage) мається на увазі метод для визначення того, які частини коду запущені протягом певного періоду часу, часто шляхом оснащення коду, який випробується, спеціальними командами, які виконуються на початку і наприкінці функції. Якщо код виконується, спеціальні команди записують те, які функції були виконані. Записування того, який код покривається тестовим випадком, можете вказати, скільки додаткового тестування необхідно, щоб здійснити покриття всього коду [3].

Зокрема, для подолання проблеми контрольної суми для Corpus Distillation розроблений метод підкомандного профілювання, тобто порівняння команд (таке як безпосередне порівняння) розбиті на шматки бітових розмірів і оцінка охоплення цих команд залежить від «глибини» порівняння. На підставі профілювання підкоманд Corpus Distillation здатний генерувати правильні контрольні суми CRC в PNG файлах без необхідності рішення обмеження. Однак через відсутність дрібнозернисті відстеження вад та ідентифікації полів контрольної суми, щоб пройти контрольну перевірку, Corpus Distillation має мутувати всі біти у вхідному зразку до досягнення полів контрольної суми, що може сильно обмежити його ефективність.

Крім того, Corpus Distillation та TaintScope можуть взаємодоповнювати один одного. Поки методи визначення розміщення контрольних точок контрольної суми та її обходу TaintScope можуть бути використані в Corpus Distillation для поліпшення методу профілювання підкоманд, метод вибору і мутації вхідного зразка на основі покриття коду в Corpus Distillation також може бути використаний у фазі фаззінгу системи TaintScope [22].

Методики аналізу протоколів та форматів вхідних даних для керівництва фаззінгом. Під протоколом у інформаційно-комунікаційних технологіях розуміють угоду або стандарт, який контролює та забезпечує комунікацію та передачу даних між двома кінцевими точками. Окремі компоненти, з яких складається протокол, зазвичай називають полями. У специфікаціях протоколів вказується, як ці поля впорядковані та відокремлені один від одного. Протоколи та формати файлів описують структуру даних і можуть мати відкриті або закриті стандарти [1].

Чим більше відомо про структуру вхідних даних, тим краще можна визначити їх ділянки, які слід обробляти для покращення ефективності фаззінгу. Якщо структура вхідних даних визначається закритим протоколом або форматом, то виникає необхідність у методах їх дослідження.

Багато засобів зворотного інжинірингу протоколів (такі як Prospex [28], Tupni [29], AutoFormat [30], Polyglot [31], Discoverer [32], FFE/x86 [33]) можуть бути використані для керівництва фаззінгом. Ці інструменти можуть витягти специфікації формату для вхідних даних на основі аналізу мережевого трафіку [32], моніторингу виконання програми під час обробки вхідних даних [29; 30; 31; 34; 35], або аналізі бінарних виконуваних кодів безпосередньо [33]. Витягнуті специфікації протоколу можуть у подальшому переведені на специфікації фаззінгу. Однак, жодна з цих систем явно не вирішує проблему перевірку контрольної суми.

У табл. 1 за результатами аналізу приведені специфічні ключові поняття, які розкривають концепцію та зміст відповідних категорій методик, які були встановлені на основі аналізу досліджень в сфері фаззінгу, наведених у списку літератури.

Висновки. За результатами проведеного аналізу відомих методик фаззінгу, усі дослідження у сфері фаззінгу структуровані на чотири узагальнюючі категорії методик. Загальний недолік традиційних методів фаззінгу полягає у тому, що обробка програмою більшості неправильних вхідних даних передчасно переривається. Щоб подолати цей недолік були розроблені інші методики фаззінгу. Для фаззінг-методик білого ящика, заснованих на символічному виконанні, характерна проблема автоматичного породження тестових випадків з правильними полями контрольних сум. Цю проблему частково вирішує методика безпосереднього фаззінгу, заснована на аналізі вад. Методики аналізу протоколів та форматів вхідних даних для керівництва фаззінгом дозволяють покращити ефективність фаззінгу, завдяки реконструкції специфікації протоколу або формату вхідних даних та визначенню для подальшої обробки певних їх ділянок. Проте остання категорія методик не розглядає проблему контрольних сум.

Також з'ясована суть специфічних ключових понять, які розкривають концепцію та зміст категорій методик фаззінгу.

Таблиця 1

Категорії методик фаззінгу та специфічні поняття, які розкривають їх концепцію

Категорії методик фаззінгу	**Приклади фаззерів, що реалізують методику**	**Специфічні ключові поняття**		
		англомовний термін	український відповідник	Номер у списку джерел
традиційний фаззінг	Spike, Peach, SNOOZE, FileFuzz	black box	чорний ящик	[1]
		data generation	породження даних	[1], [3]
		data mutation	мутація даних	[1], [3]
фаззінг-методики білого ящика, засновані на символічному виконанні	CUTE, DART, EXE, SAGE, KLEE, SmartFuzz, Replayer	white box	білий ящик	[1], [9]
		test case	тестовий випадок	[14]
методики безпосереднього фаззінгу, засновані на аналізі вад	BuzzFuzz, Corpus Distillation, TaintScope	taint checking	перевірка на вади	[18], [19], [20], [21], [22]
		code coverage	покриття коду	[3]
методики аналізу протоколів та форматів вхідних даних для керівництва фаззінгом	Prospex, Tupni, AutoFormat, Polyglot, Discoverer, FFE/x86	protocol	протокол	[1], [3]
		protocol reverser engineering	зворотній інжиніринг протоколу	[1]

Результати цієї роботи можуть бути використані при розробці узагальненої класифікації методів фаззінгу.

Список літератури

1. Саттон М. Fuzzing: Исследование уязвимостей методом грубой силы / Майкл Саттон, Адам Грин, Педрам Амини. — Пер. с англ. — СПб. : Символ-Плюс, 2009. — 560 с. — ISBN 978-5-93286-147-9.
2. Miller B. P. An empirical study of the reliability of UNIX utilities / B. P. Miller, L. Fredriksen, S. Bryan // Commun. ACM. — 1990. — №12. — P. 32–44.
3. Oehlert P. Violating assumptions with fuzzing / P. Oehlert // IEEE Security and Privacy. — 2005. — Issue 2. — P. 58–62.

4. Козиол Дж. Искусство взлома и защиты систем = The Shellcoder's Handbook / Джек Козиол, Дэвид Личфилд, Дэйв Эйтэл, Крис Энли, Синан Эрен, Нил Мехта, Рили Хассель. — Пер. с англ. — СПб. : Питер, 2006. — 416 с. — ISBN 5-469-01233-6.

5. Banks G. SNOOZE: toward a Stateful NetwOrk prOtocol fuzZEr / G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, G. Vigna // Proceedings of the Information Security Conference (ISC). — [New York, NY, USA] : Springer, 2006. — P. 343–358.

6. Sen K. Cute: A concolic unit testing engine for C / K. Sen, D. Marinov, G. Agha // ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, 2005. — P. 263–272.

7. Godefroid P. Dart: directed automated random testing / P. Godefroid, N. Klarlund, K. Sen // PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, 2005. — P. 213–223.

8. Cadar C. EXE: Automatically generating inputs of death / C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, D. R. Engler // Proceedings of the 13th ACM conference on Computer and communications security (CCS'06), 2006. — P. 322–335.

9. Godefroid P. Automated whitebox fuzz testing / P. Godefroid, M. Levin, D. Molnar // Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08). — [San Diego, CA, USA], 2008.

10. Cadar C. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs / C. Cadar, D. Dunbar, D. Engler // USENIX Symposium on Operating Systems Design and Implementation (OSDI'08). — [San Diego, CA, USA], 2008.

11. Molnar D. Dynamic test generation to find integer bugs in x86 binary Linux programs / D. Molnar, X. C. Li, D. A. Wagner // Proceedings of the 18th USENIX Security Symposium. — 2009.

12. Godefroid P. Grammar-based whitebox fuzzing / P. Godefroid, A. Kiezun, and M. Y. Levin // Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08). — [USA]: ACM, 2008.

13. Majumdar R. Directed test generation using symbolic grammars / R. Majumdar, R.-G. Xu // ESEC-FSE companion'07: The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering. — [New York, NY, USA] : ACM, 2007. — P. 553–556.

14. Test case [Electronic resource] / [Wikipedia contributors] // Wikipedia : The Free Encyclopedia. — Electronic data. — San Francisco : Wikimedia Foundation, 2010. — Mode of access: World Wide Web. — URL: http://en.wikipedia.org/w/index.php?title=Test_case&oldid=398287802. — Title from screen. — Description based on version dated 2010 November 22.

15. Newsome J. Replayer : Automatic protocol replay by binary analysis / J. Newsome, D. Brumley, J. Franklin, D. Song // Proceedings of the 13th ACM Conference on Computer and Communications Security. — 2006.

16. Sharif M. Impeding malware analysis using conditional code obfuscation / M. Sharif, A. Lanzi, J. Giffin, W. Lee // Proceedings of the 15th Annual Network and Distributed System Security Symposium. — [San Diego, CA, USA], 2008.

17. Brumley D. Automatic patch-based exploit generation is possible: Techniques and implications / D. Brumley, P. Poosankam, D. Song, J. Zheng // Proceedings of the 2008 IEEE Symposium on Security and Privacy. — 2008.

18. Taint checking [Electronic resource] / [Wikipedia contributors] // Wikipedia : The Free Encyclopedia. — Electronic data. — San Francisco : Wikimedia Foundation, 2010. — Mode of access: World Wide Web. — URL: http://en.wikipedia.org/w/index.php?title=Taint_checking&oldid=383882171. — Title from screen. — Description based on version dated 2010 September 9.

19. Sabelfeld A. Language-based information-flow security / A. Sabelfeld, A. C. Myers // IEEE Journal on Selected Areas in Communications. — 2003.

20. Terauchi T. Secure information flow as a safety problem / T. Terauchi, A. Aiken // 12th International Static Analysis Symposium. — 2005.

21. Ganesh V. Taint-based directed whitebox fuzzing / V. Ganesh, T. Leek, M. Rinard // Proceedings of the 31st International Conference on Software Engineering (ICSE'09). — [New York, NY, USA] : ACM, 2009. — P. 474–484.

22. Wang T. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection / T. Wang, T. Wei, G. Gu, W. Zou // 2010 IEEE Symposium on Security and Privacy. — P. 497-512. — DOI: 10.1109/SP.2010.37.

23. Drewry W. Flayer: Exposing Application Internals / W. Drewry, T. Ormandy // First Workshop On Offensive Technologies (WOOT). — 2007.

24. Nethercote N. Valgrind: a framework for heavyweight dynamic binary instrumentation / N. Nethercote, J. Seward // PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation. — [New York, NY, USA] : ACM, 2007. — P. 89–100.

25. Moser A. Exploring multiple execution paths for malware analysis / A. Moser, C. Kruegel, E. Kirda // SP'07: Proceedings of the 2007 IEEE Symposium on Security and Privacy. — [Washington, DC, USA] : IEEE Computer Society, 2007. — P. 231–245.

26. Wilhelm J. A forced sampled execution approach to kernel rootkit identification / J. Wilhelm, Tzi-cker Chiueh // 10th International Symposium on Recent Advances in Intrusion Detection (RAID'07), pages 219–235, 2007.

27. Ormandy T. Making Software Dumberer [Electronic resource] / T. Ormandy. — Electronic data. — [Mountain View, California, USA] : Google, 2010. — Mode of access: World Wide Web. — URL: http://taviso.decsystem.org/making_software_dumber.pdf. — Title from screen.

28. Comparetti P. M. Prospex: Protocol specification extraction / P. M. Comparetti, G. Wondracek, C. Kruegel, E. Kirda // IEEE Symposium on Security and Privacy. — [USA] : IEEE Computer Society Press, 2009.

29. Cui W. Tupni: automatic reverse engineering of input formats / W. Cui, M. Peinado, K. Chen, H. J. Wang, L. Irun-Briz // CCS '08: Proceedings of the 15th ACM conference on Computer and communications security. — [New York, NY, USA] : ACM, 2008. — P. 391–402.

30. Lin Z. Automatic protocol format reverse engineering through context-aware monitored execution / Z. Lin, X. Jiang, D. Xu, X. Zhang // Proceedings of the 15th Annual Network and Distributed System Security Symposium. — [San Diego, CA, USA], 2008.

31. Caballero J. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis / J. Caballero, H. Yin, Z. Liang, D. Song // Proceedings of ACM Conference on Computer and Communication Security. — 2007.

32. Cui W. Discoverer: Automatic protocol reverse engineering from network traces / W. Cui, J. Kannan, H. J. Wang // Proceedings of the 16th USENIX Security Symposium. — 2007.

33. Junghee J. L. Extracting output formats from executables / J. L. Junghee, T. Reps, B. Liblit // Working Conference on Reverse Engineering. — 2006. — P. 167–178.

34. Lin Z. Convicting exploitable software vulnerabilities: An efficient input provenance based approach / Z. Lin, X. Zhang, and D. Xu // Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSNDCCS 2008). — [Anchorage, Alaska, USA], 2008.

35. Wondracek G. Automatic network protocol analysis / G. Wondracek, P. M. Comparetti, C. Kruegel, and E. Kirda // 16th Network & Distributed System Security Symposium. — 2008.

Рецензент: Шелест М.Є.
Надійшла 02.11.2010