HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG
Posts and Telecommunications Institute of Technology

KIỂM THỬ XÂM NHẬP

KHOA AN TOÀN THÔNG TIN
TS. ĐINH TRƯỜNG DUY

HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG
Posts and Telecommunications Institute of Technology

KIỂM THỬ XÂM NHẬP

Phân tích mã nguồn

KHOA AN TOÀN THÔNG TIN
TS. ĐINH TRƯỜNG DUY

Biên soạn từ bài giảng: Nguyễn Ngọc Điệp, Bài giảng Kiểm thử xâm nhập, Học viện Công nghệ Bưu chính Viễn thông, 2021.

Mục lục

1. Các kỹ thuật phân tích mã nguồn
2. Phương pháp phân tích mã nhị phân
3. Kỹ thuật fuzzing

Phân tích mã nguồn

- Phân tích mã nguồn là quá trình đánh giá, kiểm tra và phân tích mã nguồn của một phần mềm hoặc ứng dụng để tìm ra các lỗ hổng bảo mật và các vấn đề khác liên quan đến mã nguồn. Phân tích mã nguồn có thể được thực hiện bằng kỹ thuật:
  - Phân tích thủ công
  - Phân tích tự động

Phân tích mã nguồn thủ công

- Sử dụng khi:
  - ứng dụng được lập trình bằng một ngôn ngữ không được hỗ trợ kiểm tra tự động
  - kiểm tra tất cả các khía cạnh của một chương trình
  - phân tích cấu trúc lập trình cực kỳ phức tạp để làm mẫu cho các công cụ phân tích tự động

Phân tích mã nguồn thủ công

- Phân tích mã nguồn thủ công là quá trình đọc và kiểm tra mã nguồn bằng tay để tìm ra các lỗi lập trình, lỗ hổng bảo mật và các vấn đề khác liên quan đến mã nguồn.
- Phân tích mã nguồn thủ công thường được thực hiện bởi các chuyên gia bảo mật hoặc những người có kinh nghiệm trong lĩnh vực phát triển phần mềm.

Phân tích mã nguồn thủ công

- Trọng tâm là dựa trên cách người dùng cung cấp dữ liệu được xử lý trong ứng dụng.
- Hiểu cách dữ liệu được gửi đến ứng dụng như thế nào, và điều gì xảy ra với dữ liệu đó.

Phân tích mã nguồn thủ công

Hạn chế:

- Phân tích mã nguồn thủ công có thể tốn nhiều thời gian và công sức, đặc biệt là đối với các dự án lớn hoặc phức tạp.
- Ngoài ra, phương pháp này cũng có thể bị giới hạn bởi khả năng của con người trong việc phát hiện các lỗi lập trình và lỗ hổng bảo mật trong mã nguồn.

Ví dụ 1

- Yêu cầu kiểm thử xâm nhập cho một ứng dụng web → phân tích mã nguồn của ứng dụng bằng cách đọc và hiểu từng dòng mã thủ công. Khi thực hiện cần tập trung vào những điểm sau đây:
  1. Tìm kiếm các lỗi bảo mật:
  2. Kiểm tra tính chính xác và hợp lệ của tham số đầu vào:
  3. Kiểm tra tính bảo mật của hệ thống:
  4. Kiểm tra tính tương thích của ứng dụng:
  5. Kiểm tra mã lược đồ:
- Sau khi đã phân tích và đánh giá mã nguồn của ứng dụng, có thể tìm thấy các lỗ hổng bảo mật hoặc các điểm yếu của ứng dụng để khắc phục và cải thiện tính bảo mật của ứng dụng.

Ví dụ 1

1. Tìm kiếm các lỗi bảo mật: cần đọc tất cả các tệp mã nguồn để tìm kiếm các lỗ hổng bảo mật, bao gồm các lỗ hổng XSS, SQL Injection, tràn bộ đệm và các lỗ hổng khác.
2. Kiểm tra tính chính xác và hợp lệ của tham số đầu vào: cần xác định các tham số đầu vào được sử dụng trong ứng dụng và kiểm tra tính chính xác và hợp lệ của chúng. Điều này giúp đảm bảo rằng ứng dụng hoạt động chính xác và tránh các lỗi bảo mật.
3. Kiểm tra tính bảo mật của hệ thống: cần phân tích mã nguồn để tìm các lỗ hổng bảo mật có thể liên quan đến hệ thống hoặc máy chủ. Điều này giúp đảm bảo rằng hệ thống hoạt động an toàn và tránh các cuộc tấn công từ phía hacker.

Ví dụ 1

4. Kiểm tra tính tương thích của ứng dụng: cần đọc mã nguồn để kiểm tra tính tương thích của ứng dụng với các trình duyệt khác nhau, các hệ điều hành khác nhau và các thiết bị khác nhau.
5. Kiểm tra mã lược đồ: cần xác định các phương thức được sử dụng để truy cập cơ sở dữ liệu và kiểm tra mã lược đồ để đảm bảo tính bảo mật của cơ sở dữ liệu.

Ví dụ 2

- Yêu cầu phân tích mã nguồn để tìm lỗi và điểm yếu của chương trình viết bằng ngôn ngữ C.
  1. Chạy chương trình và xem kết quả đầu ra.
  2. Đọc và hiểu mã nguồn.
  3. Kiểm tra tính hợp lệ của tham số đầu vào.
  4. Kiểm tra tính bảo mật của chương trình.
  5. Kiểm tra tính tương thích của chương trình
- Sau khi đã phân tích mã nguồn thủ công của chương trình C, bạn có thể tìm thấy các lỗ hổng bảo mật hoặc các điểm yếu của chương trình để khắc phục và cải thiện tính bảo mật của chương trình.

Ví dụ 2

1. Chạy chương trình và xem kết quả đầu ra. Cần chạy chương trình và kiểm tra kết quả đầu ra để biết chương trình hoạt động đúng hay sai. Nếu có lỗi, bạn cần xem thông báo lỗi để tìm hiểu nguyên nhân lỗi.
2. Đọc và hiểu mã nguồn. Cần đọc và hiểu mã nguồn của chương trình để biết cấu trúc và hoạt động của chương trình. Cần tìm hiểu về biến, hàm, lệnh điều kiện, vòng lặp, và các câu lệnh khác được sử dụng trong chương trình.
3. Kiểm tra tính hợp lệ của tham số đầu vào. Cần kiểm tra tính hợp lệ của tham số đầu vào để đảm bảo rằng chương trình sẽ không bị tấn công từ phía bên ngoài thông qua các tham số đầu vào không hợp lệ. Cần kiểm tra xem chương trình có kiểm tra giá trị nhập vào không hợp lệ hay không.

Ví dụ 2

4. Kiểm tra tính bảo mật của chương trình. Cần kiểm tra mã nguồn để xác định các lỗ hổng bảo mật, ví dụ như buffer overflow, format string vulnerability, race condition, và các lỗ hổng khác. Cần kiểm tra xem chương trình có sử dụng các hàm bảo mật như strcpy_s, strncat_s, và strcat_s hay không.
5. Kiểm tra tính tương thích của chương trình. Cần kiểm tra tính tương thích của chương trình với các phiên bản khác nhau của hệ điều hành và trình biên dịch. Cần xác định các hàm bị loại bỏ hoặc bị thay đổi trong các phiên bản khác nhau của hệ điều hành và trình biên dịch.

Phân tích mã nguồn tự động

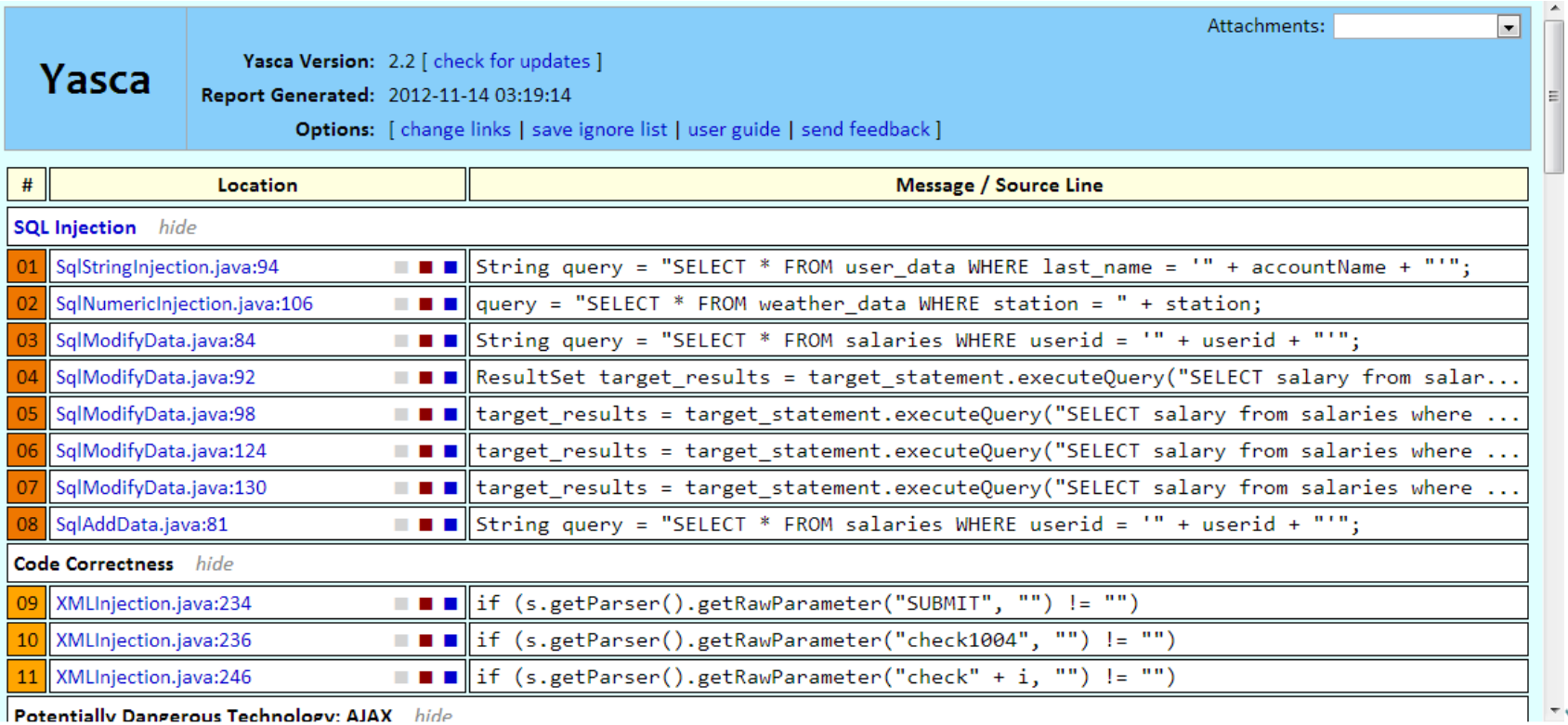
- Phân tích mã nguồn tự động là quá trình sử dụng các công cụ phân tích mã nguồn để tìm ra các lỗi lập trình, lỗ hổng bảo mật và các vấn đề khác liên quan đến mã nguồn.
- Các công cụ phân tích mã nguồn tự động có thể phân tích mã nguồn tĩnh hoặc động, tùy thuộc vào loại công cụ.

Phân tích mã nguồn tự động

- Yasca (Yet Another Source Code Analyzer) là một công cụ phân tích mã nguồn tự động mã nguồn mở được sử dụng để phát hiện các lỗ hổng bảo mật và các vấn đề khác trong mã nguồn của các ứng dụng web và phần mềm.
- Viết bằng ngôn ngữ lập trình Python và có thể chạy trên nhiều nền tảng hệ điều hành như Windows, Linux và macOS. Cho phép tự động hóa việc kiểm tra các vấn đề sau: C/C++, mã nguồn Java và class, mã nguồn JSP, PHP, Perl và Python.
- Sử dụng một loạt các plugin để phân tích mã nguồn, bao gồm các plugin để phát hiện các lỗi bảo mật như SQL injection, cross-site scripting (XSS), và các lỗ hổng bảo mật khác.

Phân tích mã nguồn tự động

- Ngoài ra, Có thể phát hiện các vấn đề khác như vi phạm quy tắc lập trình, kiểm tra chuẩn mã nguồn và phân tích khai thác mã nguồn.



Yasca

Attachments:

Yasca Version: 2.2 [check for updates]
Report Generated: 2012-11-14 03:19:14
Options: [change links | save ignore list | user guide | send feedback]

#	Location	Message / Source Line
SQL Injection *hide*		
01	SqlStringInjection.java:94	String query = "SELECT * FROM user_data WHERE last_name = '" + accountName + "'";
02	SqlNumericInjection.java:106	query = "SELECT * FROM weather_data WHERE station = " + station;
03	SqlModifyData.java:84	String query = "SELECT * FROM salaries WHERE userid = '" + userid + "'";
04	SqlModifyData.java:92	ResultSet target_results = target_statement.executeQuery("SELECT salary from salar...
05	SqlModifyData.java:98	target_results = target_statement.executeQuery("SELECT salary from salaries where ...
06	SqlModifyData.java:124	target_results = target_statement.executeQuery("SELECT salary from salaries where ...
07	SqlModifyData.java:130	target_results = target_statement.executeQuery("SELECT salary from salaries where ...
08	SqlAddData.java:81	String query = "SELECT * FROM salaries WHERE userid = '" + userid + "'";
Code Correctness *hide*		
09	XMLInjection.java:234	if (s.getParser().getRawParameter("SUBMIT", "") != "")
10	XMLInjection.java:236	if (s.getParser().getRawParameter("check1004", "") != "")
11	XMLInjection.java:246	if (s.getParser().getRawParameter("check" + i, "") != "")
Potentially Dangerous Technology: AJAX *hide*		

Phân tích mã nguồn tự động

Phân tích mã nguồn tự động có nhiều ưu điểm so với phân tích mã nguồn thủ công, bao gồm:

- Tốc độ phân tích nhanh hơn và chính xác hơn.
- Khả năng phát hiện các lỗi lập trình và lỗ hổng bảo mật nhiều hơn và chi tiết hơn.
- Giảm thiểu sự phụ thuộc vào khả năng của con người trong việc phát hiện các lỗi lập trình và lỗ hổng bảo mật.

Phân tích mã nguồn tự động

Hạn chế:

- Không thể phát hiện được tất cả các lỗi lập trình và lỗ hổng bảo mật.
- Có thể xảy ra các lỗi phát hiện sai hoặc báo động giả.
- Cần phải đầu tư một khoản tiền để sử dụng các công cụ phân tích mã nguồn tự động.

Phân tích mã chương trình đã dịch

- Phân tích mã chương trình đã dịch (hay còn được gọi là phân tích mã đối tượng) là quá trình phân tích và hiểu cấu trúc và chức năng của chương trình đã được biên dịch thành mã máy (code assembly) hoặc mã đối tượng (object code).
- Phân tích mã chương trình đã dịch thường được sử dụng để hiểu rõ hơn về cách hoạt động của chương trình, tìm ra các lỗi hoặc vấn đề bảo mật, tối ưu hóa mã và thực hiện các tác vụ khác như đảng mã, tái sử dụng mã, và tìm kiếm mã độc.
- Có 2 loại: Phân tích mã thông dịch và mã biên dịch

Phân tích mã thông dịch

- Mã thông dịch (interpreted code) là phương pháp chuyển đổi mã nguồn thành mã thực thi ngay lập tức khi chương trình đang chạy.
- Mã thông dịch được thực hiện bởi một chương trình phần mềm gọi là trình thông dịch (interpreter), trình thông dịch đọc và thực thi từng dòng mã nguồn một và trả về kết quả ngay lập tức.
- Trong quá trình này, mã nguồn được dịch thực thi theo từng dòng, nên mã nguồn và mã thực thi là khác nhau. Ví dụ Java, Python…

Phân tích mã thông dịch

- Trong trường hợp của Java, trình thông dịch này được gọi là Java Virtual Machine (JVM). Hai tính năng của byte code Java làm cho nó rất dễ dàng được dịch ngược:
  - Các tệp mã Java được biên dịch (các tệp lớp) chứa một lượng thông tin mô tả đáng kể.
  - mô hình lập trình cho JVM là khá đơn giản, và tập lệnh của nó khá ít
- Cả hai thuộc tính này đều đúng với các tập tin Python (.pyc) được biên dịch và thông dịch

Phân tích mã thông dịch

Mã nguồn gốc cho lớp PasswordChecker:

```
public class PasswordChecker {
    public boolean checkPassword(String pass) {
        byte[] pwChars = pass.getBytes();
        for (int i = 0; i < pwChars.length; i++) {
            pwChars[i] += i + 1;
        }
        String pwPlus = new String(pwChars);
        return pwPlus.equals("qcvw|uyl");
    }
}
```

Phân tích mã thông dịch

Chạy JReversePro trên tập tin được biên dịch PasswordChecker.class thu được:

```
public class PasswordChecker {
    public PasswordChecker() {
        ;
        return;
    }
    public boolean checkPassword(String string) {
        byte[] iArr = string.getBytes();
        int j = 0;
        String string3;
        for (;j < iArr.length;) {
            iArr[j] = (byte)(iArr[j] + j + 1);
            j++;
        }
        string3 = new String(iArr);
        return (string3.equals("qcvw|uyl"));
    }
}
```

Phân tích mã biên dịch

- Mã biên dịch (compiled code) là phương pháp chuyển đổi mã nguồn thành mã thực thi trước khi chương trình được chạy.
- Mã biên dịch được thực hiện bởi một chương trình phần mềm gọi là trình biên dịch (compiler), trình biên dịch dịch toàn bộ mã nguồn thành mã thực thi trước khi chương trình được chạy.
- Trong quá trình này, mã nguồn và mã thực thi đều có thể được lưu trữ trên đĩa cứng hoặc bộ nhớ. Ví dụ C, C++.
- Công cụ sử dụng cho việc dịch mã là IDA pro (Interactive Disassemble Professional

So sánh

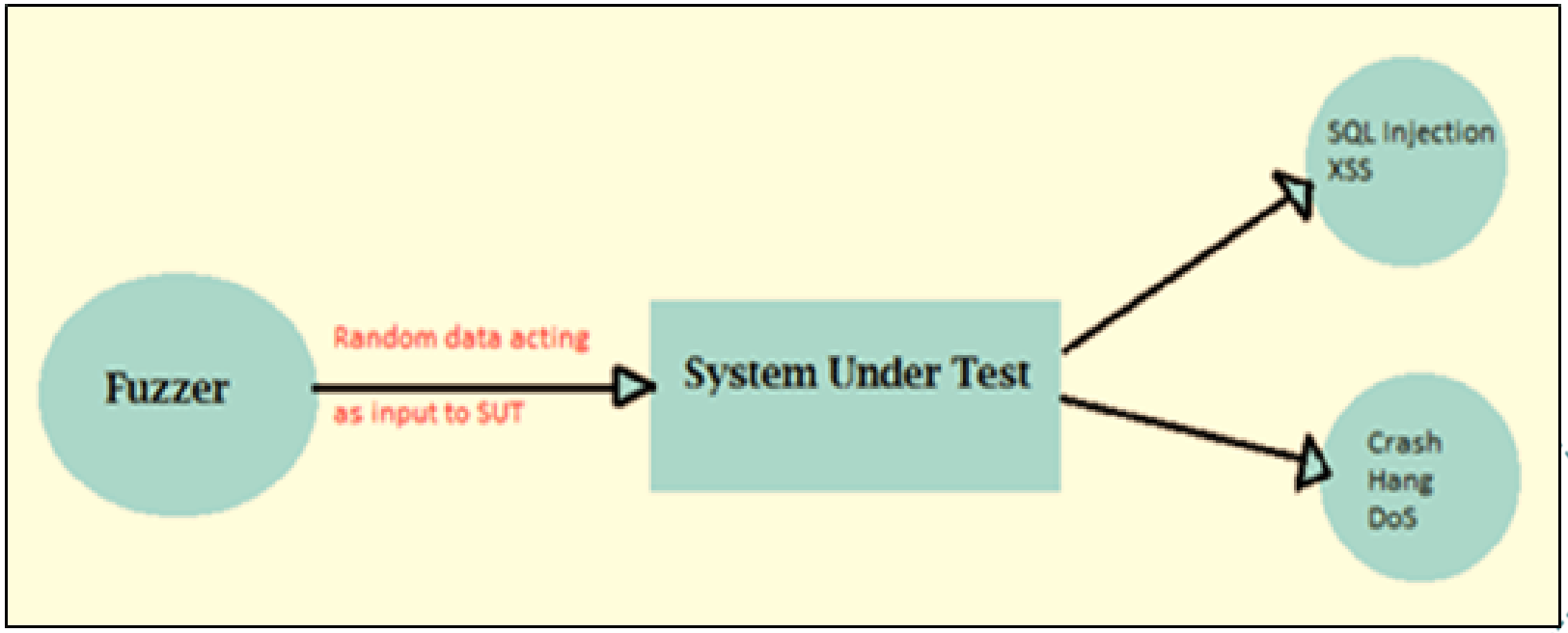
	Mã thông dịch	Mã biên dịch
Ưu điểm	- Có thể thực thi nhanh hơn vì không cần phải chuyển đổi toàn bộ mã nguồn thành mã thực thi trước khi chạy - Cho phép cập nhật mã nguồn và kiểm tra lỗi dễ dàng hơn	- Cho phép thực thi nhanh hơn trong quá trình chạy vì toàn bộ mã nguồn đã được chuyển đổi thành mã thực thi trước khi chạy. - Đảm bảo tính bảo mật cao hơn bởi vì mã nguồn đã được dịch thành mã thực thi trước khi chạy
Nhược điểm	- Có thể chậm hơn mã biên dịch trong quá trình thực thi và yêu cầu một trình thông dịch để chạy - Không đảm bảo được tính bảo mật cao bởi vì mã nguồn được dịch thực thi ngay lập tức	- Có thể yêu cầu nhiều tài nguyên hơn và cần phải biên dịch lại nếu có thay đổi trong mã nguồn

Kỹ thuật fuzzing

- Fuzzing là một kỹ thuật phát hiện lỗi phần mềm bằng cách tự động hoặc bán tự động sử dụng phương pháp lặp lại thao tác sinh dữ liệu sau đó chuyển cho hệ thống xử lý.
- Cung cấp dữ liệu đầu vào cho chương trình sau đó theo dõi và ghi lại các lỗi, các kết quả trả về của ứng dụng trong quá trình xử lý của chương trình. Kỹ thuật này được sử dụng để tìm ra các lỗi bảo mật như buffer overflow, lỗi xác thực và các lỗ hổng khác.
- Dữ liệu là ngẫu nhiên hoặc các dữ liệu không hợp lệ, dữ liệu không mong đợi: các giá trị vượt quá biên, các giá trị đặc biệt có ảnh hưởng tới phần xử lý, hiển thị của chương trình…

Kỹ thuật fuzzing

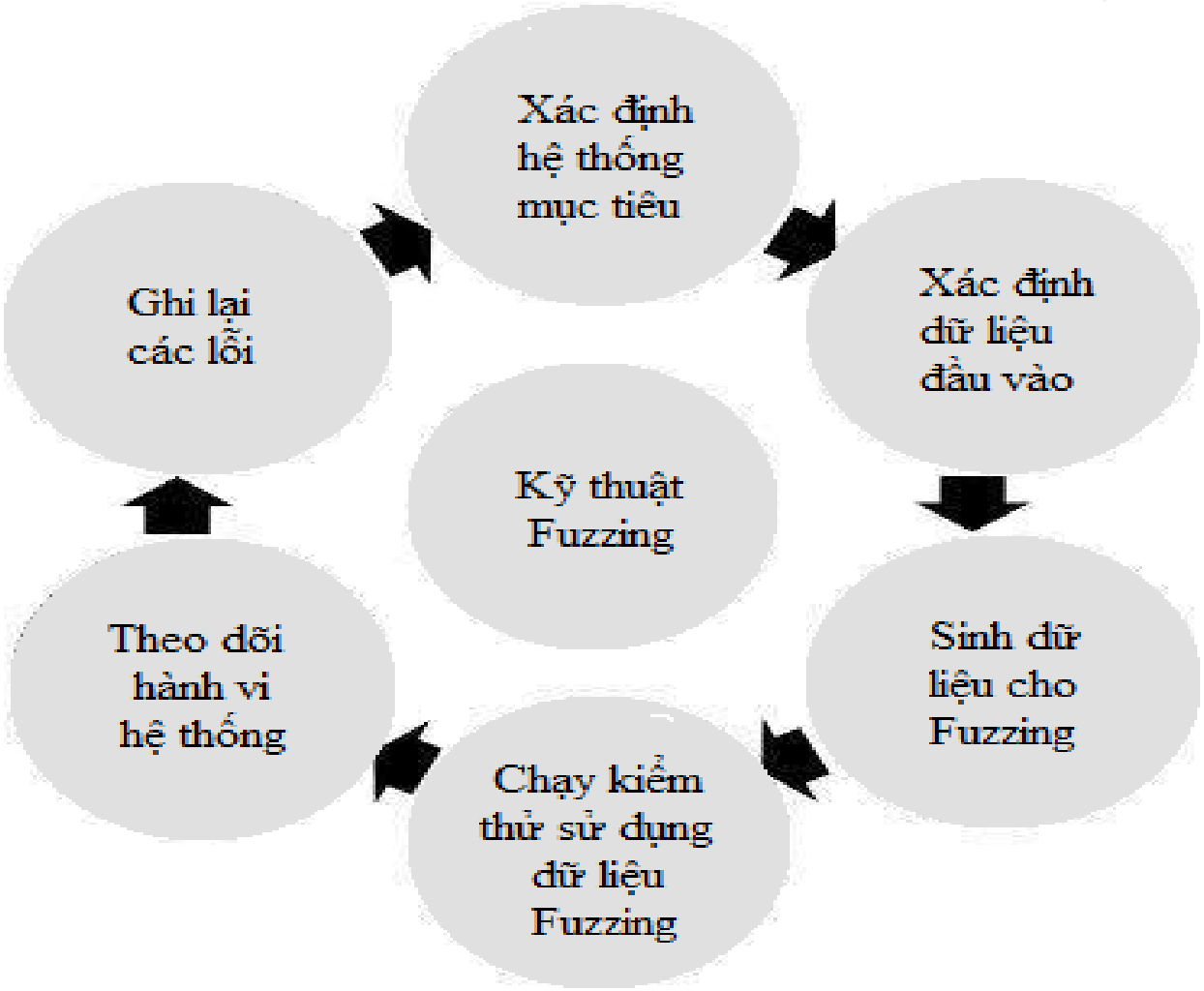
- Fuzzing là một trong những kỹ thuật kiểm thử hộp đen, không đòi hỏi quyền truy nhập vào mã nguồn
- Các chương trình và framework được dùng để tạo ra kỹ thuật fuzzing hoặc thực hiện fuzzing được gọi là Fuzzer



Kỹ thuật fuzzing

Ưu điểm	Nhược điểm
- Rất hiệu quả trong tìm kiếm những lỗi nghiêm trọng nhất về bảo mật hoặc các khiếm khuyết trong chương trình. - Cải thiện vấn đề về an ninh khi kiểm tra phần mềm - Lỗi được tìm thấy bằng fuzzing đôi khi nghiêm trọng và hầu hết là những lỗi mà tin tặc hay sử dụng: làm chương trình dừng, rò rỉ bộ nhớ, các ngoại lệ chưa kiểm soát… - Tìm ra được những lỗi không được tìm thấy khi kiểm thử bị hạn chế về thời gian và nguồn lực	- Không thể cung cấp đầy đủ về lỗi bảo mật tổng thể - Kém hiệu quả hơn với các lỗi không gây ra sự cố treo phần mềm như: virut, trojan,… - không cung cấp nhiều kiến thức về hoạt động nội bộ của các phần mềm - Với chương trình có các đầu vào phức tạp đòi hỏi phải tốn thời gian hơn để tạo ra một fuzzer đủ thông minh

Các bước thực hiện kỹ thuật Fuzzing



Fuzzing khi biết giao thức

- Bước 1: Xác định mục tiêu
- Bước 2: Xác định đầu vào
- Bước 3: Tạo dữ liệu
- Bước 4: Thực hiện test sử dụng dữ liệu fuzz
- Bước 5: Giám sát dữ liệu fuzz
- Bước 6: Xác định khả năng khai thác

Fuzzing khi biết giao thức

- **Bước 1: Xác định mục tiêu**
  - Các mục tiêu được đánh giá có nguy cơ rủi ro cao
    - Các lỗ hổng do lỗi của người lập trình hệ thống: SQL Injection
    - các lỗi do việc cấu hình hệ thống không an toàn
  - Các ứng dụng nhận dữ liệu qua mạng
  - Các ứng dụng xử lý thông tin có giá trị
  - Các ứng dụng xử lý thông tin cá nhân

Fuzzing khi biết giao thức

- **Bước 2: Xác định đầu vào**
  - Các lớp đầu vào ứng với các fuzzer phổ biến như sau:
    - Đối số dòng lệnh
    - Biến môi trường(ShareFuzz)
    - Ứng dụng Web (WebFuzz)
    - Các định dạng file (FileFuzz)
    - Giao thức mạng (SPIKE)
    - Đối tượng Memory COM (COMRaider)
    - Liên lạc liên tiến trình - IPC

Fuzzing khi biết giao thức

- **Bước 3: Tạo dữ liệu**
  - Các yêu cầu sau:
    - Tạo ra dữ liệu thử nghiệm ở các mức độ đảm bảo thỏa mãn điều kiện ứng dụng đầu vào
    - Dữ liệu được tạo ra có thể dạng file nhị phân, file văn bản được sử dụng lặp đi lặp lại vào thời điểm bắt đầu của mỗi lần kiểm tra
    - Việc tạo ra dữ liệu kiểm tra với nhiều testcase lặp đi lặp lại cung cấp các trường hợp kiểm thử để bắt lỗi khi chạy chương trình
  - Hiệu quả của fuzzing phụ thuộc vào :
    - Độ bao phủ không gian đầu vào
    - Chất lượng của dữ liệu kiểm thử

Fuzzing khi biết giao thức

- **Bước 4: Thực hiện test sử dụng dữ liệu fuzz**
  - Đối tượng tiếp cận của Fuzzing bao gồm: kiểu số, ký tự, siêu dữ liệu, các chuỗi nhị phân, định dạng file, các giao thức mạng…
  - Cách tiếp cận chung cho fuzzing là:
    - Sinh tập dữ liệu giá trị nguy hiểm được biết đến ứng từng loại đầu vào
    - Chèn thêm mã thực thi vào mã máy của chương trình
    - Phân tích hoạt động của chương trình trong lúc thực thi

Fuzzing khi biết giao thức

- **Bước 5: Giám sát dữ liệu fuzz**
  - Định nghĩa các lỗi được phát hiện
  - Có sự hiểu biết rõ về hoạt động xử lý, và có thể được tích hợp vào sự kiện phân loại lỗi tự động.

Fuzzing khi biết giao thức

- **Bước 6: Xác định khả năng khai thác**
  - Sau khi một hoặc một số lỗi phần mềm đã được xác định, các fuzzer gửi một danh sách các lỗi này cho đội ngũ phát triển.

Fuzzing khi chưa biết rõ giao thức

- Xây dựng fuzzer cho các giao thức mở là một vấn đề lớn, cả với giao thức tĩnh và giao thức động.
- Nếu không hiểu rõ về cấu trúc của giao thức, thì việc tạo ra các giá trị đầu vào ngẫu nhiên có thể dẫn đến các tình huống không mong muốn, bao gồm làm cho phần mềm bị treo hoặc gây ra các lỗi nghiêm trọng.
- Để giải quyết có thể sử dụng kỹ thuật dịch ngược theo một số mức độ hoặc sử dụng các công cụ giám sát mạng như: Wireshark… để nắm bắt tất cả lưu lượng đến và đi từ ứng dụng và hiển thị nó theo cách để cô lập dữ liệu lớp ứng dụng muốn tập trung

Phương pháp khai thác khi tìm ra lỗ hổng

- Tìm ra một vấn đề tiềm ẩn hoặc gây ra một lỗi chương trình có thể sử dụng phân tích tĩnh, phân tích động hoặc kết hợp cả 2.
- Với phân tích tĩnh, cần xác định chính xác cách tiếp cận với mã nguồn có lỗ hổng trong khi chương trình đang thực hiện.
- Sau đó cần thực hiện phân tích đông là thử nghiệm chống lại một chương trình đang chạy để xác nhận phân tích tĩnh
- Khi sử dụng fuzzer, dữ liệu fuzzer cần được chia thành các phần cần thiết cho việc trích đường dẫn mã nguồn.

Kết hợp phân tích tĩnh và phân tích động

- **Phân tích tĩnh**
  - phương pháp phân tích mã nguồn của phần mềm mà không cần thực thi chương trình.
  - Phương pháp này giúp người dùng tìm ra các lỗ hổng bảo mật, kiểm tra tính đúng đắn của mã nguồn và đưa ra các biện pháp khắc phục.
  - Các công cụ phân tích tĩnh như trình biên dịch, trình phân tích cú pháp, trình phân tích tĩnh, ... được sử dụng để thực hiện phân tích tĩnh.
  - chỉ giúp tìm ra các lỗ hổng có thể được phát hiện trong mã nguồn
- **Phân tích động**
  - phương pháp phân tích phần mềm bằng cách thực thi chương trình để tìm ra các lỗ hổng bảo mật.
  - Phương pháp này giúp người dùng tìm ra các lỗ hổng mà không cần phải có mã nguồn của phần mềm.
  - Các công cụ phân tích động như trình chụp màn hình, trình gỡ lỗi, trình kiểm tra dữ liệu đầu vào, ... được sử dụng để thực hiện phân tích động.
  - chỉ giúp tìm ra các lỗ hổng có thể được phát hiện trong quá trình thực thi chương trình

Xem xét khả năng khai thác

- **Khả năng khai thác**
  - Khả năng khai thác của một lỗ hổng phụ thuộc vào nhiều yếu tố như khả năng tương tác với phần mềm, độ phức tạp của lỗ hổng, khả năng kiểm soát luồng dữ liệu, khả năng tràn bộ đệm, ...
  - Nếu lỗ hổng có khả năng khai thác cao, người dùng có thể khai thác lỗ hổng đó để kiểm tra tính bảo mật của phần mềm.
- **Gỡ lỗi cho khai thác**
  - Sau khi xác định khả năng khai thác của lỗ hổng, người dùng cần tiến hành gỡ lỗi để tìm ra nguyên nhân gây ra lỗ hổng và đưa ra các biện pháp khắc phục.
  - Sử dụng bộ gỡ lỗi sẽ cho người dùng hình ảnh rõ ràng nhất về cách dữ liệu đầu vào của họ đã gây lỗi dừng cho một ứng dụng
  - Việc gỡ lỗi giúp nâng cao tính bảo mật của phần mềm và giảm thiểu các rủi ro bảo mật.

Xem xét khả năng khai thác

- **Phân tích ban đầu**
  - quá trình đầu tiên để xác định vị trí và nguyên nhân của lỗi trong chương trình
  - Hai câu hỏi cần được trả lời đó là tại sao chương trình bị lỗi dừng đột ngột và chương trình xảy ra sự cố ở vị trí nào.
- **Chỉ dẫn phân tích con trỏ**
  - Nếu con trỏ lệnh (eip trên x86) trỏ đến mã chương trình hợp lệ, thì lệnh trước đó là nguyên nhân của lỗi dừng chương trình.
  - Phân tích lệnh và các thanh ghi sử dụng có thể cung cấp thông tin về nguyên nhân của lỗi. Nếu eip trỏ đến vị trí bất kỳ, cần xác định xem liệu có thể chèn mã vào vị trí được tham chiếu bởi eip hay không, hoặc có thể kiểm soát nơi mà eip trỏ tới và chuyển hướng nó đến một vị trí có chứa dữ liệu do người dùng cung cấp.

Xem xét khả năng khai thác

- **Phân tích thanh ghi**
  - Quá trình xem xét các thanh ghi trong hệ thống máy tính để hiểu những thông tin quan trọng được lưu trữ trong đó và cách chúng có thể được sử dụng trong quá trình thực thi chương trình.
  - Trong quá trình phân tích, người dùng cần tìm hiểu cách mỗi thanh ghi được sử dụng để lưu trữ thông tin, bao gồm các giá trị và địa chỉ bộ nhớ để hiểu được lỗi và cách khắc phục.

Xem xét khả năng khai thác

- **Nâng cao độ tin cậy của khai thác**
  - Quá trình tìm hiểu nội dung thanh ghi trong máy tính để tìm ra địa chỉ của shellcode và thực hiện một bước nhảy gián tiếp vào shellcode → địa chỉ của shellcode được xác định chính xác và đáng tin cậy
  - Các kỹ thuật tìm kiếm địa chỉ đó trong thanh ghi bao gồm việc sử dụng tràn bộ đệm để tìm kiếm jmp esp hoặc call esp; sử dụng getopcode hoặc msfpescan để tìm kiếm một lệnh jmp hoặc call trong mã máy Linux hoặc Windows để tìm ra địa chỉ shellcode.

Tạo payload để khai thác

- **Nghiên cứu xây dựng payload:**
  - Quá trình tạo ra một đầu vào chính xác để khai thác một lỗ hổng trong chương trình. Payload có thể được hiểu là một mã độc được chèn vào các trường hoặc bộ đệm của chương trình để thực hiện các hành động xâm nhập hệ thống.
  - Quá trình tạo ra payload bao gồm các yếu tố sau:
    - Các phần tử giao thức: Các thành phần này được sử dụng để đưa các ứng dụng có lỗ hổng theo một đường dẫn chính xác.
    - Padding, NOP hoặc các phần tử khác: Được sử dụng để thay đổi các bộ đệm, tạo ra sự khác biệt giữa các đầu vào khác nhau.
    - Khai thác việc kích hoạt dữ liệu: Ví dụ như địa chỉ trả về hay địa chỉ ghi. Các thành phần này được sử dụng để khai thác lỗ hổng và thực hiện các hành động xâm nhập hệ thống.
    - Các mã thực thi: Payload/shellcode. Được sử dụng để thực hiện các hành động xâm nhập hệ thống.

Tạo payload để khai thác

- **Nghiên cứu xây dựng payload:**
  - Việc tạo ra payload đúng cách là rất quan trọng để thành công trong cuộc tấn công. Nếu đầu vào không được tạo ra đúng, công cụ khai thác sẽ không hoạt động chính xác.
  - Các vấn đề gây ra kết quả sai bao gồm
    - các thành phần giao thức sai,
    - địa chỉ trả về không khớp,
    - dữ liệu điều khiển heap sai,
    - đặt nhầm vị trí shellcode,
    - dữ liệu đầu vào chứa kí tự đặc biệt làm sai vị trí trong bộ nhớ và chương trình đích thực hiện chuyển đổi bộ đệm khiến cho shellcode bị sửa đổi.

Tạo payload để khai thác

- **Các phần tử giao thức payload**
  - Các thành phần giao thức được sử dụng để tạo payload trong quá trình khai thác lỗ hổng chương trình. Để thực hiện khai thác, cần phải biết đầy đủ các giao thức dẫn tới phần chứa lỗ hổng trong chương trình và tạo payload để chèn vào bộ nhớ của chương trình.

- **Các vấn đề liên quan tới bộ đệm**
  - Liên quan đến việc khai thác tràn bộ đệm. Khi bộ đệm bị tràn, các thông tin điều khiển trong bộ đệm sẽ bị thay đổi và người khai thác có thể nắm được quyền điều khiển.
  - Tuy nhiên, việc sắp xếp lại ngăn xếp bằng cách đặt các biến không phải là mảng giữa các bộ đệm đặt trên ngăn xếp và các địa chỉ trả về có thể gây khó khăn trong việc tạo dữ liệu đầu vào và ngăn chặn việc thực hiện thay đổi cấu trúc điều khiển.
  - Khi thực hiện khai thác tràn bộ đệm, cần cân nhắc xem có bao nhiêu biến được sử dụng và khi nào chương trình có thể bị ngừng bất thường nếu có bất kì giá trị nào bị thay đổi.

Các phương pháp phòng ngừa khai thác lỗ hổng

- ***Một số phương pháp phòng ngừa phổ biến***
  - Port knocking
  - Di chuyển
- ***Tạo bản vá***
  - Xem xét vá mã nguồn mở
  - Cân nhắc bản vá nhị phân

Một số phương pháp phòng ngừa phổ biến

- Các câu hỏi về tiêu chuẩn đánh giá rủi ro cần phải được xem xét lại.
  - Dịch vụ này có thực sự cần thiết không? Nếu không thì tắt nó đi.
  - Dịch vụ có thể được tiếp cận một cách công khai không? Nếu không, thì dùng tường lửa bảo vệ.
  - Tất cả các tùy chọn không an toàn có được tắt không? Nếu không, cần thay đổi các tùy chọn.

Một số phương pháp phòng ngừa phổ biến

- ***Port knocking***
  - áp dụng cho giới hạn người dùng để bảo vệ các dịch vụ mạng
  - sử dụng một chuỗi cổng để đóng/mở truy nhập đến dịch vụ mạng, và người dùng phải trải qua một trình tự gõ cần thiết để được kết nối
  - không phải là cơ chế bảo vệ thích hợp cho các dịch vụ truy nhập công cộng
  - Không giải quyết các lỗ hổng mà chỉ gây khó khăn hơn khi tiếp cận các lỗ hổng.

Một số phương pháp phòng ngừa phổ biến

- ***Di chuyển***
  - Phương pháp di chuyển đến một hệ điều hành mới hoặc chuyển đến một ứng dụng mới có thể giúp cải thiện hệ thống an ninh bằng cách chống lại các lỗ hổng.
  - Việc chọn lựa hệ điều hành mới hoặc ứng dụng mới phải xem xét các tính năng bảo mật của chúng và khả năng chống lại các loại khai thác lỗ hổng
  - Chuyển đến một ứng dụng mới có thể gặp nhiều thách thức như thiếu hụt các lựa chọn thay thế, di chuyển dữ liệu và tác động đến người dùng
  - Việc chuyển đổi phải được đánh giá kỹ lưỡng để đảm bảo rằng nó là lựa chọn tốt nhất để đáp ứng các lỗ hổng mới phát hiện

Tạo bản vá

- Việc tạo bản vá để vá các lỗ hổng là cách duy nhất để bảo vệ một ứng dụng và tránh khoảng thời gian dài phải đối mặt với các lỗ hổng
- Nếu không có quyền truy cập vào mã nguồn, cách đơn giản nhất là để nhà cung cấp cấp bản vá. Tuy nhiên việc này sẽ mất nhiều thời gian.
- Nếu mã nguồn là mở thì có thể thực hiện tự vá các lỗ hổng.

Vá mã nguồn mở

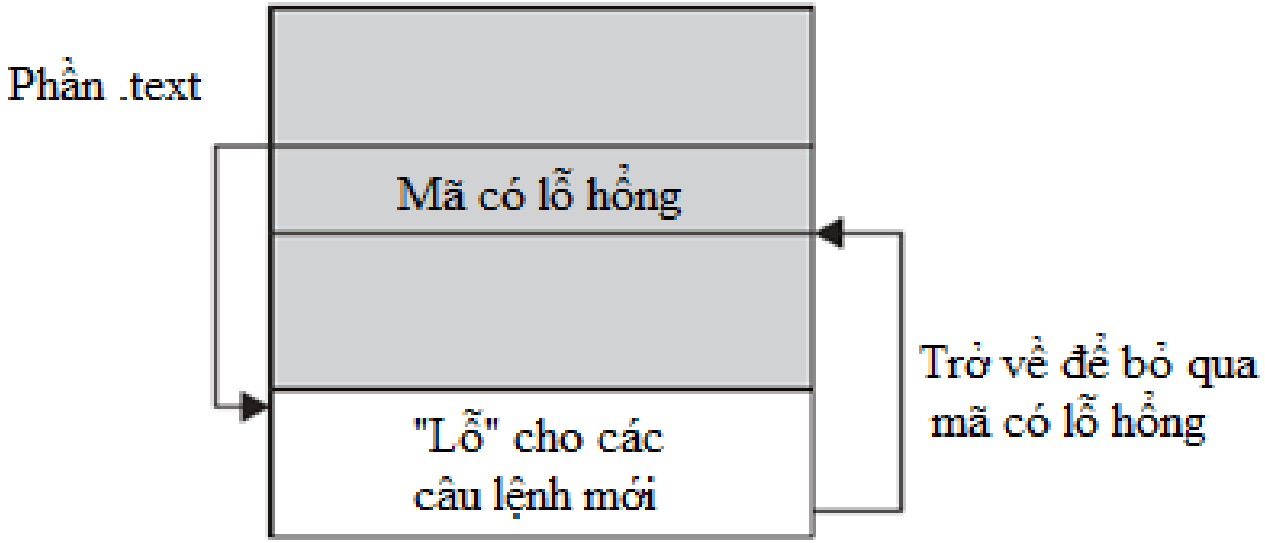
- Việc sửa lỗi trên mã nguồn mở dễ dàng hơn so với mã nguồn nhị phân, và người dùng có thể đóng góp vào việc phát triển và bảo trì mã nguồn mở.
- Tuy nhiên, việc phát triển bản vá phải đảm bảo không gây ra lỗ hổng mới và phải hiểu rõ kiến trúc của hệ thống phần mềm.
- Các công cụ diff và patch được sử dụng để tạo và áp dụng các bản vá lỗi.

Vá nhị phân

- Trong trường hợp không thể truy nhập vào mã nguồn gốc của chương trình, có thể buộc phải xem xét vá lỗi chương trình nhị phân
- Lý do: Lỗ hổng trong phần mềm không còn được nhà cung cấp hỗ trợ. Các nhà cung cấp không/chậm đưa ra các bản vá.
- Khi thực hiện vá cần phải lưu ý là bất kỳ thay đổi nào với một đoạn mã nhị phân đã biên dịch cần phải đảm bảo không chỉ hoạt động của chương trình có lỗ hổng được sửa chữa, mà còn cả cấu trúc của tệp tin nhị phân không bị hỏng.

Vá nhị phân

- Có thể sử dụng phương pháp đặt các vị trí lệnh mới (lỗ) trong không gian ảo của chương trình nhị phân để bỏ qua mã của lỗ hổng



- Cần lưu ý về vấn đề phân phối khi một bản vá nhị phân được tạo ra và thử nghiệm thành công

Tổng kết

- Các kỹ thuật phân tích mã nguồn thủ công và tự động
- Phương pháp phân tích mã nhị phân cũng như mã thông dịch của chương trình đã dịch
- Kỹ thuật fuzzing
- phương pháp khai thác khi tìm ra lỗ hổng
- phương pháp ngăn ngừa khai thác các lỗ hổng