# Chương 1: Giới thiệu môn học

- **Common Weakness Enumeration (CWE)** là danh sách các loại điểm yếu phổ biến trong phần mềm và phần cứng có liên quan đến bảo mật. Một “điểm yếu” là một điều kiện trong phần mềm, firmware, phần cứng, hoặc thành phần dịch vụ mà, dưới một số hoàn cảnh nhất định, có thể góp phần tạo ra các lỗ hổng bảo mật

- **Một lỗ hổng (vulnerability)** là một khiếm khuyết (flaw) hoặc điểm yếu (weakness) trong hệ thống máy tính, phần mềm, các quy trình bảo mật, các kiểm soát nội bộ, hoặc thiết kế và triển khai, có thể bị khai thác để vi phạm chính sách bảo mật của hệ thống

- **CVE, viết tắt của Common Vulnerabilities and Exposures**, là danh sách các lỗ hổng bảo mật máy tính đã được công bố công khai

- An toàn phần mềm là một trong những khía cạnh chính để xây dựng hệ thống/phần mềm tin cậy (trustworthy/reliability), nó là nguyên lý triển khai các cơ chế bảo mật trong việc thiết kế, xây dựng, và kiểm thử để giúp phần mềm duy trì chức năng (hoặc khả năng) chống lại các cuộc tấn công

# Chương 2: Qui trình thiết kế và phát triển phần mềm

## 1. Qui trình thiết kế và phát triển phần mềm

- Software development life cycle (SDLC) là một qui trình phát triển phần mềm, bắt đầu từ việc "lên ý tưởng" cho đến kết thúc ở khâu "phân phối phần mềm". Qui trình gồm 6 giai đoạn. Mỗi giai đoạn có đầu vào là kết quả của giai đoạn ngay trước nó

- Mô hình SDLC tạo ra phần mềm tốt hơn, nhanh hơn và ít chi phí hơn so với Waterfall ( thác nước )

### 1.1. Thu thập và phân tích yêu cầu:

- Giai đoạn thu thập và phân tích các yêu cầu bao gồm việc khám phá tình hình hiện tại, nhu cầu và ràng buộc của các bên liên quan, cơ sở hạ tầng hiện tại, v.v. và xác định vấn đề cần giải quyết bằng phần mềm

- Sau khi thu thập các yêu cầu, nhóm phân tích kết quả để xác định những điều sau:

+ Có thể phát triển phần mềm theo những yêu cầu này không và nó có thể được thực hiện với ngân sách không?

+ Có bất kỳ rủi ro nào đối với lịch trình phát triển không, và nếu có, chúng là gì?

+ Phần mềm sẽ được kiểm tra như thế nào?

+ Phần mềm sẽ được chuyển giao khi nào và như thế nào?

- Vào cuối giai đoạn này, phương pháp thác nước cổ điển đề xuất tạo tài liệu “Đặc tả yêu cầu phần mềm” (Software Requirement Specification - SRS), trong đó nêu rõ các yêu cầu và phạm vi phần mềm, đồng thời xác nhận điều này một cách tỉ mỉ với các bên liên quan => Đầu ra là Tài liệu Yêu cầu KH, Tài liệu đề xuất giải pháp (Proposal), Tài liệu Quản lý dự án (Project Plan), Tài liệu Đặc tả yêu cầu cho mục phân tích

- Tài liệu SRS (Software Requirements Specification) là một tài liệu chính thức mô tả chi tiết các yêu cầu phần mềm cho một dự án. Đây là tài liệu quan trọng trong quá trình phát triển phần mềm, giúp đảm bảo rằng các yêu cầu từ phía khách hàng hoặc người dùng được hiểu rõ và thống nhất giữa tất cả các bên liên quan, bao gồm nhóm phát triển, đội kiểm thử và khách hàng

- Do Sales, BA, PM, Customer chịu trách nhiệm cho lấy yêu cầu

- Với phân tích thì do PM, BA, Senior, Software Architect

### 1.2. Thiết kế

**-** Trong giai đoạn Thiết kế, các kiến trúc sư và nhà phát triển phần mềm thiết kế phần mềm dựa trên SRS được cung cấp

- Ở cuối giai đoạn, đội ngũ thiết kế tạo ra các tài liệu thiết kế mức cao - HighLevel Design (HLD); và thiết kế Mức Thấp - Low-Level Design (LLD) => đầu ra là Hồ sơ thiết kế, Flowchart, Diagram …

- Thiết kế cấp cao hay HLD đề cập đến hệ thống tổng thể, một thiết kế bao gồm mô tả về kiến trúc và thiết kế hệ thống; và là một thiết kế hệ thống chung bao gồm:

+ Kiến Trúc Hệ Thống

+ Thiết kế cơ sở dữ liệu

+ Mô tả ngắn gọn về hệ thống, dịch vụ, nền tảng và mối quan hệ giữa các môđun

- Thiết kế cấp thấp (LLD) đề cập đến quá trình xác định thiết kế chức năng, chi tiết của hệ thống hoặc thành phần phần mềm. Nó liên quan đến việc xác định các mô-đun riêng lẻ, cấu trúc dữ liệu, thuật toán, giao diện và đầu vào/đầu ra của hệ thống. Nó bao gồm:

+ Data Flow Design

+ Data Structure Design

+ Algorithm Design

+ Component-Level Design

+ Architectural Design

- Mục đích của thiết kế cấp thấp là cung cấp một mô tả rõ ràng và chính xác về cách hệ thống sẽ hoạt động và cách các thành phần khác nhau của nó sẽ tương tác với nhau

- Do Designer, BA chịu trách nhiệm

### 1.3. Phát triển/Hiện thực

- Vì tất cả các thành phần và mô-đun được xây dựng trong giai đoạn này, nên đó là giai đoạn dài nhất của vòng đời

- Vào cuối giai đoạn, mã chức năng (functional code) thực hiện tất cả các yêu cầu của khách hàng đã sẵn sàng để được kiểm tra => đầu ra là source code

- Do Developers chịu trách nhiệm

### 1.4. Kiểm thử

- Thực hiện kiểm thử mã nguồn đã phát triển trong môi trường kiểm thử

- Bao gồm các dạng kiểm thử: Functional testing, integration testing, performance testing and security testing

- Quá trình kiểm tra tiếp tục cho đến khi tất cả các mã đều không có lỗi và vượt qua tất cả các bài kiểm tra. Vào cuối giai đoạn này, một phần mềm chất lượng cao, không có lỗi, đã sẵn sàng để đưa ra sử dụng (production) => đầu ra: Test case, Test plan, Bug Report …

- Do QA, Tester thực hiện

### 1.5. Triển khai

- Phần mềm được cài đặt trên môi trường sản phẩm sẽ chạy thực tế (production environment)

- Vào cuối giai đoạn, người quản lý sản phẩm phát hành phần mềm cuối cùng cho người dùng cuối => đầu ra là tài liệu Đặc tả hệ thống, tài liệu Hướng dẫn sử dụng, tài liệu Cấu hình và cài đặt

- Do DevOps chịu trách nhiệm

### 1.6. Bảo trì

- Trong suốt giai đoạn Bảo trì, đội ngũ sẽ:

+ Cung cấp sự hỗ trợ cho khách hang

+ Gỡ lỗi được tìm thấy trên sản phẩm phần mềm

+ Cải tiến phần mềm

+ Thu thập thông tin mới từ khách hang

- Cuối cùng, nhóm sẽ làm việc trên vòng lặp tiếp theo của SDLC và phiên bản tiếp theo của phần mềm

- Do DevOps chịu trách nhiệm chính

## 2. Phương pháp luận phát triển phần mềm

- Phương pháp luận phát triển phần mềm (software development methodology) thường được biết đến với thuật ngữ mô hình Chu kỳ Phát triển phần mềm (Software Development Life Cycle)

- Có 3 phương pháp phát triển phần mềm phổ biến (SDLC):

+ Traditional SDLC: Waterfall

+ Agile SDLC: Agile scrum

+ Agile SDLC: Lean

- Việc sử dụng các phương pháp phát triển phần mềm, phụ thuộc vào:

+ Loại dự án (project)

+ Độ dài/thời gian thực hiện dự án (length of project)

+ Số lượng nhân lực của đội ngũ phát triển phần mềm (size of team)

### 2.1. Phương pháp phát triển phần mềm Waterfall ( truyền thống )

- Mô hình nguyên bản của Waterfall gồm 7 giai đoạn:

A diagram of a program

Description automatically generated

- Khi nào nên chọn mô hình phát triển phần mềm truyền thống (Plan-driven):

+ Các giai đoạn và yêu cầu được xác định rõ ràng

+ Đưa ra sản phẩm mẫu chạy thử (protype) là một việc khó

+ Biết trước tất cả các yêu cầu trước khi bắt đầu dự án

+ Việc thay đổi không phải là ưu tiên

+ Các vai trò (role) được định nghĩa chặt chẽ

+ Được khuyến nghị sử dụng trong các dự án phát triển với thời gian dài, chi phí cao

+ Tài liệu hóa chi tiết, toàn bộ quá trình phát triển

### 2.2. Phương pháp phát triển phần mềm agile

- Phương pháp Agile là phương pháp phát triển phần mềm tập trung vào tính linh động và hướng khách hàng (customer-focused)

- Theo Agile Manifesto, giá trị của phương pháp Agile bao gồm:

+ Chú trọng vào qui trình làm việc và tương tác của thành viên hơn là công cụ phần mềm hỗ trợ

+ Phần mềm hoạt động như yêu cầu hơn là chú trọng vào tài liệu thiết kế

+ Việc cộng tác với khách hàng được thực hiện qua các hợp đồng thoả thuận

+ Đáp ứng sự thay đổi so với việc tuân theo một kế hoạch

- Các Phương pháp Agile phổ biến:

+ Agile Scrum: Scrum tập trung vào các nhóm nhỏ, tự tổ chức, nhóm họp hàng ngày trong thời gian ngắn và làm việc trong các cuộc chạy nước rút (sprint) lặp đi lặp lại

* Nhóm Scrum có khả năng làm việc phối hợp (crossfunctional), hợp tác, tự quản lý và tự trao quyền
* Các nhóm scrum không được lớn hơn 10 cá nhân. Một Scrum Team điển hình thì nên trong khoảng từ 3-9 người, bao gồm cả Product Owner và Scrum Master
* Scrum master nên có một cuộc họp trực tiếp hàng ngày với nhóm vào một thời gian cố định hàng ngày, không quá 15 phút để "điểm lại" các nhiệm vụ (task) quan trọng đã hoàn thành, đang thực hiện hoặc sắp bắt đầu

+ Lean: Phương pháp Lean nhấn mạnh vào việc loại bỏ nỗ lực hao phí trong việc lập kế hoạch và thực hiện, đồng thời tối đa hoá giá trị đối với khách hàng. Qui tắc Lean bao gồm 7 bước:

* Giảm thiểu hao phí (Eliminate waste): xác định và triệt tiêu hao phí trong qui trình SDLC
* Tăng cường việc học tập (Amplify learning): thông qua việc thực hành; thông qua sự tương tác và tích lũy kinh nghiệm từ những gì khách hàng muốn/thích.
* Quyết định càng muộn càng tốt (Decide as late as possible)
* Phân phối càng nhanh càng tốt (Deliver as fast as possible)
* Trao quyền cho nhóm (Empower the team): Tạo sự hợp tác làm việc phối hợp từ bên trong nhóm hơn là từ người quản lý cấp trên
* Xây dựng sự toàn vẹn từ bên trong (Build integrity in)
* Tối ưu hóa toàn bộ (Optimize the whole): suy nghĩ cho thời gian dài hạn, ưu tiên Khách hàng > Côngty > Nhóm làm việc > Cá nhân

+ Extreme Programming (XP): XP hướng tới giải quyết các loại vấn đề cụ thể về chất lượng mà các nhóm phát triển phần mềm phải đối mặt

2.2.1. Một vài khái niệm cơ bản:

- **Sprint** là một khoảng thời gian cụ thể, thường từ 2-4 tuần, trong đó, mỗi nhóm đảm nhận nhiều nhiệm vụ (còn được gọi là User Story - câu chuyện của người dùng) mà họ cảm thấy có thể hoàn thành. Khi sprint kết thúc, phần mềm sẽ hoạt động và có thể phân phối được. • Khoảng thời gian của sprint được xác định trước khi quá trình bắt đầu và hiếm khi thay đổi

- **Công việc tồn đọng (BackLog)** bao gồm tất cả các tính năng của phần mềm, nằm trong danh sách ưu tiên

- **Câu chuyện người dùng (user story)** là một tuyên bố đơn giản về những gì người dùng (hoặc một vai trò) cần và tại sao. Mỗi câu chuyện của người dùng phải đủ nhỏ để một nhóm có thể hoàn thành nó trong một lần chạy nước rút (sprint)

## 3. Mẫu thiết kế phần mềm

- Mẫu thiết kế phần mềm - Software design patterns là các giải pháp để giải quyết các vấn đề thường gặp trong phát triển phần mềm, sử dụng chúng có thể đẩy nhanh quá trình phát triển phần mềm

- Các mẫu thiết kế không phụ thuộc vào ngôn ngữ lập trình, tức:

+ Nó ưu tiên thành phần đối tượng hơn là kế thừa lớp

+ Được lập trình theo giao diện, không phải triển khai

### 3.1. Mẫu thiết kế Observer Design Pattern

- Mẫu thiết kế observer là mẫu theo dạng đăng ký – thông báo (subscription - notification), cho phép các đối tượng nhận các sự kiện khi có thay đổi ở một đối tượng mà chúng đang quan sát

- Hiện thực cơ chế đăng ký:

+ Đối tượng cần có khả năng lưu trữ các đối tượng đang theo dõi nó (các observer)

+ Đối tượng cần có phương thức để thêm và xóa các observer

- Ưu điểm của thiết kế Observer: observer có thể nhận được theo thời gian thực khi có thay đổi xảy ra

- Cơ chế đăng ký thường cho hiệu suất tốt hơn các lựa chọn khác, ví dụ cơ chế khảo sát (polling)

### 3.2. Mẫu thiết kế MVC

- Mẫu Model-View-Controller (MVC) nhằm đơn giản hóa việc phát triển các ứng dụng có giao diện người dùng

- MVC trừu tượng hóa mã nguồn (code) và chức năng thành 3 thành phần khác nhau:

+ Model: Cấu trúc dữ liệu của ứng dụng và có nhiệm vụ quản lý dữ liệu, logic và các quy tắc của ứng dụng, nhận input từ controller

+ View: Biểu diễn trực quan của dữ liệu

+ Controller: đóng vai trò trung gian giữa model và view, nhận input từ người dùng và thay đổi nó để phù hợp với định dạng cho model hoặc view

- Ưu điểm của MVC: mỗi thành phần có thể được build song song

## 4. Hệ thông quản lí phiên bản

- Tên gọi: Version control, version control systems, revision control,source control

- Là phương pháp để quản lý các thay đổi trong các tập tin để lưu trữ lịch sử thay đổi

- Lợi ích:

+ Cho phép phối hợp làm việc

+ Hỗ trợ truy vết và trực quan (Accountability and visibility)

+ Làm việc độc lập

+ Đảm bảo an toàn

+ Làm việc từ bất kỳ đâu

- Có 3 loại hệ thống quản lí phiên bản:

+ Cục bộ - Local

+ Tập trung – Centralized

+ Phân tán – Distributed

### 4.1. Hệ thống quản lý phần mềm cục bộ - Local Version Control System (LVCS)

- LVCS sử dụng cơ sở dữ liệu đơn giản để lưu vết tất cả các thay đổi trên tập tin

- Thông thường, hệ thống lưu trữ các khác biệt (delta) giữa 2 phiên bản của tập tin

- Khi cần khôi phục phiên bản cũ, khác biệt được, các khác biệt được đảo ngược để quay lại phiên bản được yêu cầu

### 4.2. Hệ thống quản lý phiên bản tập trung Centralized Version Control System (CVCS)

- CVCS sử dụng mô hình server-client.

- Kho lưu trữ được đặt trên 1 vị trí trung tâm, là 1 server.

- CVCS chỉ có 1 cá nhân được làm việc trên 1 file nhất định tại 1 thời điểm.

- Một cá nhân cần kiểm tra để khóa tệp và thực hiện các thay đổi cần thiết, sau đó check in khi hoàn thành

### 4.3. Hệ thống quản lý phiên bản phân tán Distributed Version Control System (DVCS) – Điển hình là Git

- DVCS là mô hình ngang hàng peer-to-peer.

- Kho lưu trữ nằm trên hệ thống client, nhưng thường được lưu trong dịch vụ hosting kho lưu trữ.

- Trong DVCS, mọi cái nhân có thể làm việc trên bất kỳ file nào, bất kỳ lúc nào, do chỉ có bản sao file ở cục bộ đang được sửa đổi. Do đó, không cần phải khóa file.

- Khi cá nhân đã thay đổi xong, họ đẩy file lên kho lưu trữ chính trong dịch vụ hosting kho lưu trữ, và hệ thống quản lý phiên bản kiểm tra xem có xung đột nào giữa các thay đổi hay không

## 5. Đánh giá và kiểm thử mã nguồn

- Đánh giá mã nguồn - code review: lập trình viên xem xét các mã nguồn, 1 phần mã nguồn, hoặc các thay đổi nhất định trong mã nguồn và đưa ra phản hồi. Những lập trình viên như vậy được gọi là reviewers

- Quá trình đánh giá mã nguồn chỉ diễn ra sau khi các thay đổi trong mã nguồn đã hoàn tất và kiểm tra xong

- Mục tiêu của đánh giá mã nguồn nhằm đảm bảo mã nguồn cuối cùng:

+ Dễ đọc

+ Dễ hiểu

+ Tuân theo các phương pháp lập trình tốt nhất

+ Dùng đúng định dạng

+ Không có lỗi

+ Được chú thích và tài liệu đầy đủ

+ Sạch

### 5.1. Các loại đánh giá mã nguồn

- Đánh giá mã nguồn dạng formal: Các lập trình viên tổ chức các buổi họp để đánh giá toàn bộ mã nguồn

- Đánh giá mã nguồn theo các thay đổi: cũng được biết đến là đánh giá mã nguồn có công cụ hỗ trợ, đánh giá các mã nguồn bị thay đổi do lỗi, user story, tính năng, commit, v.v..

- Đánh giá mã nguồn dạng Over-theshoulder: Một reviewer giám sát lập trình viên viết mã nguồn và đưa ra các phản hồi

- Gửi email: các email tự động được gửi bởi các hệ thống quản lý mã nguồn khi có thay đổi

### 5.2. Kiểm thử phần mềm

- Kiểm thử phần mềm thường được chia làm 2 loại chính:

+ Kiểm thử chức năng - Functional testing kiểm tra xem phần mềm có hoạt động chính xác hay không. Phần mềm có hoạt động đúng như luồng logic đã định nghĩa, từ các cấp độ chi tiết thấp nhất với Kiểm thử Đơn vị, đến mức độ phức tạp nhất trong Kiểm thử Tích hợp?

+ Kiểm thử phi chức năng - Non-functional testing kiểm tra tính khả dụng, hiệu suất, tính bảo mật, khả năng phục hồi, tuân thủ các quy ước,... và nhiều vấn đề khác. Dạng kiểm thử này kiểm tra xem phần mềm có phù hợp với mục đích của nó, mang đến những giá trị lợi ích như mong đợi và giảm thiểu các rủi ro hay không?

- Lập trình viên xem các yêu cầu thiết kế như những bài kiểm thử và viết các phần mềm để vượt qua các bài kiểm thử đó. Đây gọi là Phát triển hướng kiểm thử (Test-Driven Development - TDD)

### 5.3. Kiểm thử đơn vị

- Kiểm thử đơn vị - Unit Testing: kiểm thử chức năng một cách chi tiết từng đoạn mã nguồn nhỏ (từng dòng, từng khối, các hàm, các class, và các thành phần độc lập khác)

- Các framework kiểm thử là các phần mềm cho phép đưa ra các nhận định có thể kiểm thử được và xác định xem các nhận định đó có hợp lệ tại 1 thời điểm thực thi hay không

### 5.4. Kiểm thử tích hợp

- Kiểm thử tích hợp đảm bảo tất cả các đơn vị riêng biệt phù hợp với nhau để tạo thành ứng dụng hoàn chỉnh.

### 5.5. Phát triển hướng kiểm thử - Test-Driven Development (TDD)

- Khi muốn kiểm tra để xác nhận thiết kế ứng dụng đáp ứng yêu cầu, nên viết các mã kiểm thử trước khi viết mã nguồn cho ứng dụng

- Sau khi biểu đạt các yêu cầu trong các mã kiểm thử, có thể viết các mã nguồn ứng dụng đến khi nào vượt qua được các phép kiểm thử đã tạo trong mã kiểm thử

- Mô hình cơ bản của TTD là một quy trình 5 bước được lặp đi lặp lại:

+ Tạo phép kiểm thử mới.

+ Chạy phép kiểm thử để tìm các lỗi không mong muốn

+ Viết mã ứng dụng để vượt qua phép kiểm thử mới

+ Chạy phép kiểm thử để tìm lỗi

+ Tái cấu trúc và cải thiện mã ứng dụng

## 6. Các định dạng dữ liệu

- Rest APIs cho phép người dùng trao đổi thông tin với các dịch vụ hoặc thiết bị ở xa

- 3 định dạng chuẩn phổ biến nhất để trao đổi thông tin với API ở xa gồm: XML, JSON, và YAML

- Phân tích định dạng XML, JSON, hay YAML là yêu cầu thường xuyên phải thực hiện khi tương tác với các API. Ví dụ về các hoạt động trong mô hình hiện thực REST API như sau:

+ Chứng thực, thường dùng phương thức POST với cặp username/password và nhận một token dùng cho chứng thực các yêu cầu tiếp theo

+ Thực hiện yêu cầu GET đến 1 đầu cuối (chứng thực khi cần thiết) để nhận trạng thái của 1 tài nguyên, với yêu cầu output dưới dạng XML, JSON, hay YAML

+ Thay đổi XML, JSON, hay YAML được trả về

+ Thay đổi XML, JSON, hay YAML được trả về

A black screen with white text

Description automatically generated

# Chương 3: SSDLC

- SSDLC (Secure Software Development Life Cycle) là một phiên bản mở rộng của SDLC (Software Development Life Cycle) với sự tập trung đặc biệt vào bảo mật. Cả hai đều là các quy trình giúp tổ chức phát triển phần mềm một cách có hệ thống và hiệu quả, nhưng SSDLC bổ sung thêm các bước hoặc yếu tố nhằm đảm bảo rằng bảo mật được tích hợp vào mọi giai đoạn phát triển phần mềm

## 1. Security Requirement

- Xác định các yêu cầu về vấn đề bảo mật thông qua các câu hỏi:

+ Hệ thống có chứa hoặc tiếp cận với các loại dữ liệu bí mật, nhạy cảm, hay dữ liệu định danh cá nhân (PII)?

+ Dữ liệu được lưu trữ ở đâu và như thế nào? Ứng dụng sẽ được công khai (qua internet) hay sử dụng nội bộ (intranet)?

+ Ứng dụng này có thực hiện các tác vụ nhạy cảm hay quan trọng không (ví dụ chuyển tiền, mở khóa, hay vận chuyển thuốc)?

+ Hệ thống cần đảm bảo tính sẵn sàng ở mức độ nào?

+ …

## 2. Secure Design

- Phần mềm được thiết kế an toàn ngay từ ban đầu

- Có tính đến tránh các hoạt động độc hại có thể xảy ra

- Nhằm giảm thiểu ảnh hưởng khi phát hiện lỗ hổng hoặc người dung nhập input không hợp lệ

**1.2.1. Các vấn đề cơ bản về bảo mật**

- Giải quyết vấn đề ở giai đoạn càng về sau trong SDLC thì chi phí bỏ ra càng lớn

- Cần đảm bảo security ngay từ khi bắt đầu dự án, sử dụng nguyên tắc Pushing Left/Shifting Left tức dịch về giai đoạn trước đó của mỗi giai đoạn để đảm bảo đã giải quyết security

- Vấn đề ở giai đoạn thiết kế: Design Flaw

+ Lỗi trong thiết kế phần mềm

+ Cho phép user thực hiện các hành động vốn không được thực hiện

+ Biện pháp: dùng security design concepts, requirements và threat modeling

- Vấn đề ở giai đoạn phát triển: Security Bugs

+ Vấn đề trong hiện thực phần mềm, trong lập trình

+ Cho phép user dùng ứng dụng một cách độc hại hoặc không chính xác

+ Biện pháp: đánh giá mã nguồn, kiểm thử security, training và hướng dẫn về secure code

**1.2.2. OWASP SAMM**

- Software Assurance Maturity Model (SAMM) là một Framework mở hỗ trợ hình thành và hiện thực chiến lược bảo mật phần mềm phù hợp với các rủi ro mà tổ chức phải đối mặt

- SAMM hỗ trợ:

+ Đánh giá các phương pháp bảo mật phần mềm hiện có của tổ chức với các mức độ và tiêu chí có sẵn

+ Thiếp lập chương trình đảm bảo bảo mật phần mềm cân bằng trong các vòng lặp phát triển

+ Demo các giải pháp cải tiến cụ thể đối với chương trình bảo mật

+ Định nghĩa và đo lường các hoạt động liên quan đến bảo mật trong tổ chức

**1.2.3. Làm sao để tránh Insecure Design**

- Tạo các user story tốt

+ Cần các yêu cầu chức năng và phi chức năng về bảo mật

+ User story: mô tả ngắn gọn, dễ hiểu của 1 chức năng phần mềm dưới góc nhìn của người dùng cuối

+ Trong user story cũng cần ghi những flaw có thể xảy ra của phần mềm

- Yếu tố security trong quy trình phát triển: Phát triển phần mềm an toàn: tính đến an toàn phần mềm ngay từ đầu và có kiểm thử bảo mật tích hợp

- Bảo mật và tách biệt các layer, thư viện: Đảm bảo các layer (ứng dụng và mạng) tách biệt rõ ràng và sử dụng các thư viện design pattern an toàn

**1.2.4. Concept**

- Protecting Sensitive Data - Bảo vệ các dữ liệu nhạy cảm

- Never Trust, Always Verify/Zero Trust/Assume Breach - Không tin bất kỳ ai, luôn phải kiểm tra

- Backup and Rollback - Sao lưu và cho phép khôi phục

- Server-Side Security Validation - Xác thực bảo mật ở phía server

- Framework Security Features - Các tính năng bảo mật của framework

- Security Function Isolation - Cô lập các hàm bảo mật

- Application Partitioning - Chia nhỏ ứng dụng

- Secret Management (not mean the secrets that users store inside the software) - Quản lý các bí mật

- Re-authentication for Transactions (Avoiding CSRF) - Xác thực lại các giao dịch (tránh CSRF)

**1.2.5. Dữ liệu trong ứng dụng**

- Production data: dữ liệu mà phần mềm sử dụng khi chạy thực tế

+ Không nên dùng khi kiểm thử, phát triển phần mềm hay bất kỳ mục đích nào khác mục đích của tổ chức

+ Phát triển và kiểm thử nên dùng dữ liệu đã che bớt thông tin (masked – anonymized)

+ Dữ liệu thật chỉ nên dùng khi chạy thực tế

- Bảo vệ mã nguồn - “Security through obscurity”

## 3. Thread Modeling – Mô hình hóa mối đe dọa

- Threat modeling — quy trình xác định các mối đe dọa tiềm ẩn đối với ứng dụng và đảm bảo các biện pháp giảm thiểu thích hợp đã được áp dụng

- Evil brainstorming – Xem xét phần mềm dưới góc nhìn của kẻ tấn công

+ Đưa ra các câu hỏi và đảm bảo đã cân nhắc các trường hợp xấu có thể xảy ra → concern

+ Đánh giá khả năng, ảnh hưởng → loại bỏ các concern không thể xảy ra hoặc không ảnh hưởng

+ Đánh giá (rate) các concern còn lại ở các mức độ cao, trung bình, thấp

→ Danh sách các threats (các concern đã được xác nhận)

→ Danh sách các risk (đánh giá cao/trung bình/thấp)

- Một số phương pháp mô hình hóa mối đe dọa: STRIDE, PASTA, TRIKE, DREAD, VAST, OCTAVE

A diagram of a flower

Description automatically generated with medium confidence

A close-up of a diagram

Description automatically generated

A diagram with blue and white text

Description automatically generated with medium confidence

A diagram with text and icons

Description automatically generated with medium confidence

A screen shot of a computer

Description automatically generated

## 4. Secure Code

- Lựa chọn framework và ngôn ngữ lập trình

+ Sử dụng framework đang và sẽ được hỗ trợ trong thời gian dài

+ Chọn phiên bản mới nhất hoặc mới-thứ-hai

+ Hỗ trợ security → có đáp ứng được các tính năng cần thiết?

- HTTP methods: tất cả các method HTTP không sử dụng thì nên disable để giảm khả năng tấn công

- Identity – Quản lý định danh: Không nên tự tạo hệ thống riêng, nên sử dụng các giải pháp có sẵn

- Authentication và Authorization – Chứng thực và Phân quyền truy cập

- …

**1.4.1. Thiết lập secure coding baselines**

- Secure coding baselines là các yêu cầu secure code tối thiểu và danh sách checklist để team dự án chuyển sang giai đoạn tiếp theo

- Cũng là 1 phần trong các điều kiện cần đảm bảo để sản phẩm có thể được phân phối. Tất cả các dự án phải được quét với công cụ quét mã nguồn nhất định trước khi phân phối, ví dụ: kiểm tra các cảnh báo, lỗi

- Bên cạnh đó, secure coding baselines cần các công cụ phát triển có liên quan và hoạt động training trong thực thế thay vì chỉ là các rule secure code trong các tài liệu

**1.4.2. Training về Mã nguồn an toàn**

- Mục đích: thông báo cho đội ngũ phát triển biết về các phương pháp secure code sẽ được áp dụng

- Ở giai đoạn đầu của training, tập trung vào:

+ Chuẩn hay secure code baseline là gì?

+ Các vấn đề trong secure code thường thấy trong doanh nghiệp?

+ Chúng sẽ tác động như thế nào đến công việc của lập trình viên?

+ Các tiêu chí để quét mã nguồn an toàn?

=> Thay vì các rule về lập trình an toàn đơn thuần, tốt hơn hết nên đưa ra các case study hoặc ví dụ về mã nguồn có lỗ hổng trong 1 ngữ cảnh nào đó

**1.4.3. Lựa chọn công cụ đánh giá**

- Các tiêu chí lựa chọn công cụ:

+ Dễ sử dụng

+ Chi phí thấp

+ Hỗ trợ ngôn ngữ lập trình

+ Tỉ lệ phát hiện và tỉ lệ dương tính giả

+ Cập nhật các rule quét mã

- Có 2 cách quét mã nguồn:

+ Quét mã nguồn tĩnh với IDE plugin, tương tự như công cụ kiểm tra chính tả, cú pháp, giúp lập trình viên học và sửa các vấn đề bảo mật một cách trực quan

+ Quét mã nguồn hàng ngày để tạo các báo cáo quét hàng ngày, lập trình viên xem các báo cáo quét mã hàng ngày để khắc phục hoặc nhận xét các vấn đề bảo mật theo đợt

## 5. Secure Testing & Deployment

**1.5.1. Kiểm thử hộp trắng / hộp đen**

- Kiểm thử hộp trắng là phương pháp kiểm thử phần mềm mà người kiểm thử có quyền truy cập và hiểu rõ cấu trúc, logic, và mã nguồn của ứng dụng, ta sẽ kiểm tra các điều kiện logic, các luồng dữ liệu, các cấu trúc vòng lặp và điều kiện và đảm bảo bao phủ đầy đủ mã lệnh (Code Coverage)

- Kiểm thử hộp đen là phương pháp kiểm thử mà người kiểm thử không cần biết về cấu trúc bên trong hoặc mã nguồn của ứng dụng, ta kiểm tra tính đúng đắn của đầu ra dựa trên các giá trị đầu vào cụ thể và thực hiện các kịch bản kiểm thử (test cases) với dữ liệu đầu vào khác nhau

**1.5.2. Kiểm thử an toàn với SDLC**

A diagram of security testing

Description automatically generated

## 6. DevOps

- Là sự kết hợp giữa phát triển (development) và vận hành (operations) phần mềm, là văn hóa làm việc trong các công ty sử dụng quy trình phát triển Agile

- Phương pháp mới như Continuous Integration, Continuous Delivery and Continuous Deployment cùng với sự phát triển của DevOps, tập trung vào:

+ Tương tác, phối hợp và gắn kết giữa các team

+ Áp dụng các phương pháp để tự động hóa thay đổi, cấu hình và triển khai

+ Chuyển giao giải pháp nhanh hơn

+ Theo dõi và lập kế hoạch cập nhật sản phẩm nhanh chóng

- Lợi ích của DevOps:

+ Cải thiện sự hợp tác, hỗ trợ chức năng và khắc phục lỗi nhanh hơn

+ Tăng tính linh hoạt, nhanh chóng và tin cậy

+ Bảo mật cơ sở hạ tầng và bảo vệ dữ liệu

+ Bảo trì và cập nhật nhanh hơn

+ Chuyển đổi các dự án với chiến lược số hóa

+ Tăng tốc độ, năng suất của team kinh doanh và CNTT

# Chương 4: Phân tích - Kiểm thử phần mềm & Fuzzing cơ bản

## 1. Kiểm thử phần mềm

- Kiểm thử phần mềm là quá trình đánh giá và xác minh xem phần mềm hoặc ứng dụng có hoạt động đúng như mong đợi hay không

- Mục tiêu chính của kiểm thử phần mềm:

+ Phát hiện và sửa chữa các lỗi

+ Đảm bảo chất lượng sản phẩm

+ Thẩm định và xác minh rằng phần mềm đáp ứng đúng các yêu cầu đặt ra

+ Cải thiện độ tin cậy của phần mềm

- Có nhiều cách phân loại Kiểm thử, như sau:

+ Phân loại theo cách thức kiểm thử

+ Phân loại theo giai đoạn kiểm thử

+ Phân loại theo mục tiêu kiểm thử

+ Phân loại theo tự động hóa kiểm thử

### 1.1. Phân loại theo cách thức kiểm thử

- Kiểm thử hộp đen (Black Box Testing):

+ Tập trung vào đầu vào và đầu ra của hệ thống mà không cần quan tâm đến cách thức hoạt động bên trong.

+ Mục tiêu là kiểm tra chức năng của phần mềm dựa trên các yêu cầu, mà không biết về mã nguồn.

- Kiểm thử hộp trắng (White Box Testing):

+ Dựa vào kiến thức về mã nguồn của hệ thống để kiểm thử các cấu trúc và logic bên trong của chương trình.

+ Mục tiêu là kiểm tra các đường dẫn, điều kiện và cách thức xử lý của mã nguồn.

- Kiểm thử hộp xám (Gray Box Testing):

+ Là sự kết hợp giữa kiểm thử hộp đen và hộp trắng, người kiểm thử có kiến thức một phần về hệ thống bên trong và sử dụng kiến thức này để kiểm tra các phần cụ thể

### 1.2. Phân loại theo giai đoạn kiểm thử

- Kiểm thử đơn vị (Unit Testing): Kiểm tra từng thành phần hoặc đơn vị nhỏ nhất của mã nguồn (thường là các hàm hoặc lớp) để đảm bảo chúng hoạt động đúng.

- Kiểm thử tích hợp (Integration Testing): Kiểm thử sự tương tác giữa các thành phần hoặc module đã được kết hợp lại với nhau để đảm bảo rằng chúng hoạt động chung một cách mượt mà.

- Kiểm thử hệ thống (System Testing): Kiểm tra toàn bộ hệ thống để đảm bảo rằng các thành phần làm việc cùng nhau và phần mềm hoàn chỉnh đáp ứng đúng yêu cầu.

- Kiểm thử chấp nhận (Acceptance Testing): Kiểm thử ở giai đoạn cuối để xác minh rằng phần mềm đã đáp ứng đúng yêu cầu của khách hàng và có thể triển khai

### 1.3. Phân loại theo mục tiêu kiểm thử

- Kiểm thử chức năng (Functional Testing): Tập trung vào việc kiểm tra các chức năng của phần mềm, đảm bảo rằng nó thực hiện đúng các yêu cầu được xác định.

- Kiểm thử phi chức năng (Non-functional Testing): Kiểm tra các yếu tố phi chức năng của phần mềm, như hiệu suất, bảo mật, độ tin cậy, khả năng sử dụng.

- Kiểm thử hồi quy (Regression Testing): Được thực hiện sau khi có sự thay đổi trong mã nguồn để đảm bảo rằng các tính năng cũ vẫn hoạt động bình thường và không bị ảnh hưởng bởi sự thay đổi.

- Kiểm thử bảo mật (Security Testing): Kiểm tra khả năng bảo vệ hệ thống và dữ liệu khỏi các mối đe dọa bảo mật như tấn công xâm nhập, lỗ hổng bảo mật, và các cuộc tấn công mạng

- Kiểm thử hiệu suất (Performance Testing):

+ Kiểm tra xem hệ thống hoạt động nhanh như thế nào dưới các điều kiện tải khác nhau.

+ Bao gồm các loại như kiểm thử tải (load testing), kiểm thử stress (stress testing), kiểm thử khả năng mở rộng (scalability testing).

- Kiểm thử khả năng sử dụng (Usability Testing): Đánh giá phần mềm từ góc nhìn của người dùng, kiểm tra mức độ dễ sử dụng, giao diện thân thiện, và trải nghiệm người dùng

### 1.4. Phân loại theo tự động hóa kiểm thử

- Kiểm thử thủ công (Manual Testing):

+ Người kiểm thử thực hiện các kịch bản kiểm thử theo cách thủ công mà không sử dụng công cụ tự động.

+ Phù hợp với các trường hợp kiểm thử giao diện người dùng và trải nghiệm người dùng.

- Kiểm thử tự động (Automated Testing):

+ Sử dụng công cụ và kịch bản tự động để thực hiện kiểm thử, giúp tăng tốc độ kiểm thử và giảm thiểu lỗi do con người.

+ Phù hợp với các loại kiểm thử lặp đi lặp lại như kiểm thử hồi quy và kiểm thử hiệu suất

- Kế hoạch kiểm thử hay kế hoạch đảm bảo chất lượng gồm:

+ Mục đích của đảm bảo chất lượng dự án

+ Những tài liệu liên quan có thể tham khảo

+ Việc quản lý chất lượng dự án được tiến hành như thế nào

+ Các chuẩn, các luật được đặt ra cho việc lập trình hoặc kiểm thử, cấu hình, đơn vị đo lường cho chất lượng và việc thực hiện các kiểm thử

+ Quản lý những rủi ro của dự án: gắn phần rủi ro của đảm bảo chất lượng dự án với bản kế hoạch quản lý toàn bộ các rủi ro của dự án

### 1.5. Qui trình kiểm thử

- Quy trình kiểm thử phần mềm (Software Testing Life Cycle - STLC) không có tiêu chuẩn cố định, về cơ bản bao gồm các giai đoạn:

A white background with black text

Description automatically generated

## 2. Chiến lược Kiểm thử Hộp trắng

- Thiết kế test case dựa vào cấu trúc nội tại bên trong của đối tượng cần kiểm thử

- Đảm bảo tất cả các câu lệnh, các biểu thức điều kiện bên trong chương trình đều được thực hiện ít nhất một lần

- Các kỹ thuật kiểm thử hộp trắng:

+ Basis Path Testing

+ Control-flow/Coverage Testing

+ Data-flow Testing

## 3. Kiểm thử đường dẫn cơ sở

- Kiểm thử đường dẫn cơ sở (Basis Path Testing) được McCabe đưa ra vào năm 1976

- Là phương pháp thiết kế test case đảm bảo rằng tất cả các đường dẫn độc lập (independent path) trong một code module đều được thực thi ít nhất một lần.

- Independent path: là bất kỳ path nào trong code mà bổ sung vào ít nhất một tập các lệnh xử lý hay một biểu thức điều kiện (Pressman 2001)

- Cho biết số lượng test case tối thiểu cần phải thiết kế khi kiểm thử một code module

## 4. Kiểm thử luồng điều khiển

### 4.1. Các cấp bao phủ điều khiển

- Phủ kiểm thử (coverage): tỉ lệ các thành phần thực sự được kiểm thử so với tổng thể các thành phần

- Các thành phần bao gồm: lệnh thực thi, điểm quyết định, điều kiện con hay sự kết hợp của chúng

- Độ phủ kiểm thử càng lớn thì độ tin cậy càng cao

- Có 5 cấp độ phủ:

+ Phủ cấp 0: kiểm thử những gì có thể kiểm thử được, phần còn lại để người dùng phát hiện và báo lại sau. Đây là hình thức kiểm thử không có trách nhiệm

+ Phủ cấp 1: Bao phủ câu lệnh (statement coverage): Các câu lệnh được thực hiện ít nhất 1 lần

+ Bao phủ nhánh (branch coverage): tại các điểm quyết định thì các nhánh đều được thực hiện ở cả hai phía True, False

+ Phủ cấp 3: Bao phủ điều kiện (condition coverage): Các điều kiện con của các điểm quyết định được thực hiện ít nhất 1 lần

+ Phủ cấp 4: Kết hợp phủ nhánh và điều kiện (branch & condition coverage)

### 4.2. Control-flow/Coverage Testing

A screen shot of a computer code

Description automatically generated

- Là kỹ thuật thiết kế các test case đảm bảo phủ hay “cover” được tất cả các câu lệnh, biểu thức điều kiện trong code module cần kiểm thử

- Đây là đoạn code mẫu ví dụ để xét 4 tiêu chí sau:

A math equations on a white background

Description automatically generated

- Có 4 tiêu chí đánh giá độ bao phủ:

+ Method Coverage (phương thức)

* Tỷ lệ phần trăm các phương thức trong chương trình được gọi thực hiện bởi các kịch bản kiểm thử
* Kịch bản kiểm thử cần phải đạt được 100% độ phủ phương thức (method coverage)
* Test case: foo(0,0,0,0,0) đạt 100%

+ Statement Coverage (Câu lệnh)

* Tỷ lệ phần trăm các câu lệnh trong chương trình được gọi thực hiện bởi các test case
* Test case foo(0,0,0,0,0) đạt 42% ( 5 câu lệnh được thực hiện )
* Test case foo(1,1,1,1,1) đạt 100%

+ Decision/Branch Coverage (biểu thức điều kiện)

* Tỷ lệ phần trăm các biểu thức điều kiện trong chương trình được ước lượng giá trị trả về (True, False) khi thực thi các test case
* Một biểu thức điều kiện (cho dù là đơn hay phức hợp) phải được kiểm tra trong cả hai trường hợp giá trị của biểu thức là True hay False
* Đối với các hệ thống lớn, độ phủ thường đạt dao động từ 75%-85%
* Test case foo (0,0,0,0,0) => điều kiện a=0 trả về true
* Test case foo (1,1,1,1,1) => điều kiện a = 0 trả về false, điều kiện a = b trả về true
* => phải thêm test case 3: foo(1,2,1,2,1) → để phủ100% mức điều kiện rẽ nhánh

+ Condition Coverage (biểu thức điều kiện đơn)

* Tỷ lệ phần trăm các biểu thức điều kiện đơn trong biểu thức điều kiện phức của chương trình được ước lượng giá trị trả về (true, false) khi thực hiện các test case
* Ví dụ:

A table with numbers and symbols

Description automatically generated with medium confidence

### 4.3. Data-flow Testing: Kiểm thử luồng dữ liệu

A diagram of a graph

Description automatically generated

- Là một kỹ thuật kiểm thử phần mềm, tập trung vào luồng dữ liệu (Data-Flow) trong chương trình để xác định các lỗi tiềm ẩn liên quan đến việc sử dụng và thao tác dữ liệu

- Data Flow Graph (DFG) đóng vai trò quan trọng trong việc hình dung và phân tích luồng dữ liệu, giúp phát hiện các vấn đề trong quá trình kiểm thử, là kỹ thuật thiết kế test case dựa vào việc khảo sát sự thay đổi trạng thái trong chu kỳ sống của các biến trong chương trình

- Lợi ích của kiểm thử luồng dữ liệu:

+ Tìm một biến được sử dụng nhưng chưa bao giờ được định nghĩa

+ Tìm một biến được định nghĩa nhưng chưa bao giờ được sử dụng

+ Tìm một biến được định nghĩa nhiều lần trước khi nó được sử dụng

+ Giải phóng một biến trước khi nó được sử dụng

- Ứng dụng của DFG:

+ Phân tích định nghĩa và sử dụng biến (DefinitionUse Analysis): DFG giúp theo dõi luồng của các biến từ điểm mà chúng được gán giá trị đến điểm mà chúng được sử dụng

+ Phát hiện các lỗi về sử dụng bất hợp lệ của biến: DFG giúp phân tích các đường dẫn từ điểm gán giá trị đến điểm sử dụng của biến, phát hiện khi một biến được sử dụng không đúng cách

+ Kiểm thử các đường dẫn có liên quan đến biến (Variable Path Testing): DFG giúp tạo ra các trường hợp kiểm thử dựa trên các đường dẫn từ điểm định nghĩa biến đến tất cả các điểm sử dụng của nó trong chương trình

A table with a number of letters and numbers

Description automatically generated with medium confidence

A diagram of a flowchart

Description automatically generated

## 5. Chiến lược kiểm thử hộp đen

- Kiểm thử hộp đen (black-box testing) còn gọi kiểm thử dựa trên đặc tả (specification-based testing) vì thông tin duy nhất làm cơ sở để kiểm thử hộp đen là bảng đặc tả yêu cầu chức năng của từng thành phần phần mềm

- Tester tập trung vào những gì phần mềm làm được (WHAT), không quan tâm nó làm như thế nào (HOW)

- Không cần truy cập mã nguồn

- Tách biệt khung nhìn của người dùng và lập trình viên (mô tả tình huống trong thực tế)

- Nhiều người có thể tham gia kiểm thử

### 5.1. Nhược điểm

- Kiểm thử khó đạt hiệu quả cao

- Khó thiết kế kịch bản kiểm thử (test case)

- Khó kiểm thử phủ hết được mọi trường hợp

- Không có định hướng kiểm thử rõ rang

### 5.2. Quá trình kiểm thử hộp đen

- Phân tích, xem xét các đặc tả chức năng của phần mềm

- Thiết kế kịch bản kiểm thử

- Thực thi kịch bản kiểm thử

- So sánh kết quả đạt được với kết quả mong muốn trong từng kịch bản kiểm thử

- Lập báo cáo kết quả kiểm thử

### 5.3. Các kỹ thuật kiểm thử hộp đen

- Phân vùng tương đương (Equivalence Partitioning hoặc Equivalence Class)

+ Ý tưởng của kỹ thuật này nếu một giá trị đại diện trong nhóm đúng thì các giá trị còn lại trong nhóm cũng đúng và ngược lại

+ Phương pháp phù hợp cho các bài toán có giá trị đầu vào là một miền xác định

+ Phân vùng tương đương (EP) là phân chia một tập các điều kiện kiểm thử thành các tập con có các giá trị tương đương nhau và kiểm thử các tập con này. Hai giá trị tương đương theo một trong các cách sau

* Chúng tương tự nhau (intuitive similarity).
* Tài liệu đặc tả mô tả chương trình sẽ xử lý chúng theo cùng một cách thức (specified as equivalent).
* Chúng lái chương trình theo cùng đường (chẳng hạn cùng nhánh if) (equivalent path).
* Chúng cho cùng kết quả với những giả thiết đưa ra (riskbased)

+ Mục đích của EP là giảm thiếu số lượng test case không cần thiết khi kiểm thử

+ EP có thể áp dụng tất cả cấp độ kiểm thử

A white background with black text

Description automatically generated

- Phân tích giá trị biên (Boundary Value Analysis - BVA)

+ Phân tích giá trị biên (BVA) là kỹ thuật kiểm thử dựa các giá trị tại biên giữa các phân vùng tương đương, bao gồm trường hợp:

* Hợp lệ (valid)
* Không hợp lệ (invalid)

+ Thường được áp dụng đối với các đối số của một phương thức

+ BVA hiệu quả nhất trong trường hợp “các đối số đầu vào (input variables) độc lập với nhau và mỗi đối số đều có một miền giá trị hữu hạn”

A diagram of a graph

Description automatically generated

- Bảng quyết định (Decision Table / Cause Effect)

- Kiểm thử Dịch chuyển trạng thái (State Transition Testing)

## 6. Fuzzing

### 6.1. Định nghĩa

- Fuzzing là một kỹ thuật dựa trên việc gửi các giá trị ngẫu nhiên hay rác tới một ứng dụng hoặc một tính năng/hàm của chương trình ứng dụng nhất định để quan sát hành vi bất thường của ứng dụng.

+ Fuzzing là một trong những kỹ thuật của kiểm thử hộp đen, không đòi hỏi quyền truy cập vào mã nguồn

+ Fuzzing có thể sử dụng trong kiểm thử hộp trắng, trong trường hợp có thể truy cập vào mã nguồn

- Fuzzing là một kỹ thuật phát hiện lỗi của 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

+ Nó cung cấp dữ liệu đầu vào cho chương trình (là 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), 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

- Fuzzing những ứng dụng có thể sử dụng. Một số ứng dụng phổ biến bao gồm:

+ Các trình duyệt web/ email client

+ Trình đọc ảnh

+ Bộ chuyển đổi định dạng tập tin (file format)

+ Các dịch vụ mạng (network service)

- Cách cơ bản nhất của fuzzing là gọi hàm với tất cả khả năng có thể có của dữ liệu đầu vào

### 6.2. Phân loại

- Mức độ hiểu biết về chương trình mục tiêu:

+ Black box

+ White box

+ Grey box

- Chiến lược tạo dữ liệu:

+ Mutation based

+ Generation based

- Sử dụng thông tin thu được của lần thử trước đó:

+ Phản hồi (feedback)

+ Không phản hồi (no-feedback)

### 6.3. Quy trình tổng quan

A diagram of a software process

Description automatically generated

A diagram of a software system

Description automatically generated

6.4. Các kỹ thuật Fuzzing

- Bao gồm 3 kỹ thuật chính:

+ Kỹ thuật sinh mẫu (sample generation technique)

* Kỹ thuật sinh mẫu (sample generation technique) bao gồm:
  + Biến đổi ngẫu nhiên (random mutation/blackbox fuzzing)
  + Biểu diễn cấu trúc ngữ pháp (grammar representation)
  + Các thuật toán lập lịch (scheduling algorithms)

+ Kỹ thuật phân tích động (dynamic analysis technique)

* Kỹ thuật phân tích động (dynamic analysis technique) bao gồm:
  + Dynamic symbolic execution
  + Coverage feedback
  + Dynamic taint analysis

+ Kỹ thuật phân tích tĩnh (static analysis technique)

* Kỹ thuật phân tích tĩnh (static analysis technique) yêu cầu truy cập mã nguồn. Kỹ thuật này bao gồm:
  + Control-flow analysis
  + Data-flow slices

### 6.4. Ưu điểm

- Kiểm thử fuzzing giúp tìm thấy những lỗi nghiêm trọng nhất về bảo mật hoặc khiếm khuyết của phần mềm.

- Lỗi được tìm thấy bằng fuzzing đôi khi nghiêm trọng và hầu hết là những lỗi mà tin tặc hay sử

dụng. Trong đó có crashes, rò rỉ bộ nhớ, unhandled exception, v.v.

- Những lỗi không được tìm thấy khi kiểm thử bị hạn chế về thời gian và nguồn lực thì cũng được kiểm thử fuzzing tìm ra.

- Kết quả sử dụng kiểm thử Fuzzing hiệu quả hơn khi sử dụng các phương pháp kiểm thử khác

### 6.5. Nhược điểm

- Chỉ riêng kiểm thử Fuzzing thì không thể xử lý hết được các mối đe dọa an ninh tổng thể hoặc các lỗi.

- Kiểm thử Fuzzing kém hiệu quả với các lỗi mà không gây treo chương trình, chẳng hạn như một số loại virus, computer Worm, Trojan, vv. Chỉ hiệu quả trong các tình huống cụ thể.

- Fuzzing 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

### 6.6. Tương lai của kiểm thử Fuzzing

- Trong tương lai, sẽ có nhiều công cụ fuzzing tích hợp AI và ML trong nỗ lực để làm cho các công cụ sử dụng đơn giản hơn và thông minh hơn.

- Tuy nhiên, có những lo ngại rằng tin tặc cũng sẽ thấy dễ dàng hơn khi sử dụng các công cụ này

trong việc phát hiện ra các điểm yếu bảo mật của các phần mềm trên quy mô lớn

- Ngoài ra, các doanh nghiệp sẽ yêu cầu các công c fuzzing nhanh hơn và sâu hơn để thực hiện các bài kiểm tra toàn diện trong khung thời gian ngắn nhất

### 6.7. Symbolic Excution for Fuzzing

- Symbolic Execution (Thực thi ký hiệu/tượng trưng) là một kỹ thuật phân tích chương trình, trong đó các giá trị đầu vào của chương trình không được coi là các giá trị cụ thể mà là các ký hiệu đại diện cho một tập hợp các giá trị có thể có

- Thay vì chạy chương trình với các giá trị cụ thể, symbolic execution theo dõi đường đi của chương trình với các biểu thức logic dựa trên các giá trị đầu vào ký hiệu này

+ Điều này cho phép phân tích toàn bộ không gian đầu vào của chương trình một cách có hệ thống và phát hiện lỗi hoặc lỗ hổng bảo mật

+ Khi chương trình thực thi với symbolic execution, các biến được gán với các biểu thức ký hiệu thay vì các giá trị cụ thể

+ Tại mỗi điều kiện rẽ nhánh (như if, else, hoặc while), symbolic execution xác định điều kiện nào cần thỏa mãn để tiếp tục thực hiện con đường đó

+ Kỹ thuật này sau đó sẽ tạo ra một tập hợp các điều kiện, gọi là path conditions (điều kiện đường đi), đại diện cho các điều kiện cần thỏa mãn để đi theo một nhánh cụ thể trong chương trình

\* Lợi ích của Symbolic Excution

- Phát hiện lỗi bảo mật: Bằng cách kiểm tra các đường đi có điều kiện phức tạp trong chương trình, symbolic execution có thể phát hiện các lỗi hoặc lỗ hổng mà chỉ bằng các kiểm thử thông thường có thể không phát hiện ra

+ Điều này đặc biệt hữu ích trong việc kiểm tra các chương trình có logic phức tạp hoặc chương trình an toàn như smart contracts

- Tìm ra các tình huống vi phạm: Symbolic execution có thể tìm các đường đi mà trong đó một điều kiện cụ thể gây ra lỗi (ví dụ: chia cho 0, truy cập bộ nhớ ngoài phạm vi, hoặc lỗ hổng bảo mật như buffer overflow)

\* Tại sao kết hợp Symbolic Execution vào Fuzzing?

- Symbolic Execution giúp phát hiện và tạo các trường hợp đầu vào có khả năng cao gây lỗi, từ đó hướng dẫn Fuzzing khám phá các vùng mã ít được kiểm tra

- Tối ưu hóa quá trình fuzzing bằng cách tập trung vào các đường đi quan trọng thay vì chỉ dựa vào ngẫu nhiên

- Cách hoạt động của Symbolic Execution-guided Fuzzing:

+ Symbolic Execution phân tích chương trình và tạo ra các path conditions (điều kiện đường đi).

+ Giải bài toán điều kiện: Một công cụ giải (solver) được sử dụng để tìm các giá trị đầu vào tương ứng với các điều kiện đường đi.

+ Fuzzing sử dụng các giá trị đầu vào này để kiểm thử các trường hợp cụ thể và tìm lỗi

A screenshot of a computer code

Description automatically generated

\* Lợi ích của Symbolic Execution-guided Fuzzing:

- Tăng phạm vi bao phủ của kiểm thử: Khám phá nhiều đường đi trong chương trình, bao gồm cả các đường ít khi được kích hoạt.

- Tìm lỗi sâu hơn trong chương trình: Phát hiện các lỗi logic và bảo mật khó tìm với các phương pháp Fuzzing ngẫu nhiên.

- Tối ưu hóa việc sinh dữ liệu: Giảm thiểu số lượng các trường hợp kiểm thử không hiệu quả

\* Hạn chế và thách thức cho việc kết hợp:

- Hạn chế của Symbolic Execution:

+ Dễ gặp phải vấn đề về “path explosion” (số lượng đường đi tăng quá nhiều)

+ Khả năng giải các điều kiện phức tạp có thể đòi hỏi tài nguyên tính toán lớn

- Thách thức của việc kết hợp với Fuzzing: Cân bằng giữa khả năng sinh dữ liệu hiệu quả và việc xử lý quá tải khi số lượng điều kiện đường đi quá lớn

# Chương 5: Introduction to Exploitation

## 1. Các khái niệm cơ bản

- Vulnerability – Lỗ hổng (danh từ): một lỗi trong bảo mật của hệ thống, có thể cho phép kẻ tấn công sử dụng hệ thống theo 1 cách khác với thiết kế của hệ thống. Bao gồm các vấn đề:

+ Tác động đến tính sẵn sàng của hệ thống

+ Leo thang quyền

+ Điều khiển toàn bộ hệ thống trái phép

+ …

- Exploit – khai thác/cách thức khai thác

+ (động từ) lợi dụng 1 lỗ hổng để khiến hệ thống phản ứng lại theo 1 cách khác với thiết kế ban đầu

+ (danh từ):công cụ, tập các lệnh, hay mã nguồn được dùng để lợi dụng 1 lỗ hổng. Tên gọi khác Proof of Concept (POC)

- Fuzzer (danh từ): một công cụ hoặc ứng dụng có chức năng thử tất cả, hoặc 1 loạt các đầu vào không mong muốn cho một hệ thống. Mục đích của fuzzer là xác định hệ thống có bug hay không, để tránh bị khai thác mà không cần biết tất cả về hoạt động bên trong của hệ thống

- 0day (danh từ): 1 cách thức khai thác 1 lỗ hổng chưa được tiết lộ công khai. Đôi khi cũng được dung cho khái niệm lỗ hổng. Một lỗ hổng zero-day là 1 lỗi, điểm yếu hoặc bug có trong phần mềm, firmware hoặc phần cứng đã được tiết lộ công khai nhưng chưa được vá. Các nhà nghiên cứu đã tiết lộ về lỗ hổng, và các nhà sản xuất và phát triển đã biết về vấn đề bảo mật đó, nhưng chưa có bản vá hoặc bản cập nhật chính thức để giải quyết

- 1day (danh từ): Một khi lỗ hổng đã được công bố công khai và các nhà sản xuất hay phát triển đã có các bản vá cho nó, lỗ hổng trở thành lỗ hổng đã biết hay n-day

- Khi hacker hoặc các tác nhân đe dọa phát triển và triển khai các PoCs – cách thức tấn công hoặc 1 mã độc thành công để khai thác lỗ hổng khi nhà sản xuất vẫn đang cố gắng vá lỗ hổng (hoặc đôi khi, không biết về sự tồn tại của lỗ hổng), được gọi là zero-day exploit hoặc tấn công zero-day

## 2. Assembly

- Khai thác các lỗ hổng bảo mật cần nắm chắc được hợp ngữ - assembly, vì hầu hết các cách thức khai thác (exploit) đều cần viết (hoặc chỉnh sửa) mã nguồn dạng hợp ngữ (IA32)

### 2.1. Quản lí bộ nhớ

- Khi thực thi chương trình, các thành phần khác nhau của chương trình sẽ được ánh xạ vào bộ nhớ một cách có tổ chức

- Đầu tiên, hệ điều hành tạo 1 vùng địa chỉ để chạy chương trình. Vùng địa chỉ này bao gồm các lệnh thực thi của chương trình cũng như các dữ liệu cần thiết

- Tiếp theo, các thông tin được đưa từ file thực thi của chương trình sang vùng địa chỉ vừa tạo. Có 3 dạng segment khác nhau bao gồm: .text, .bss, and .data

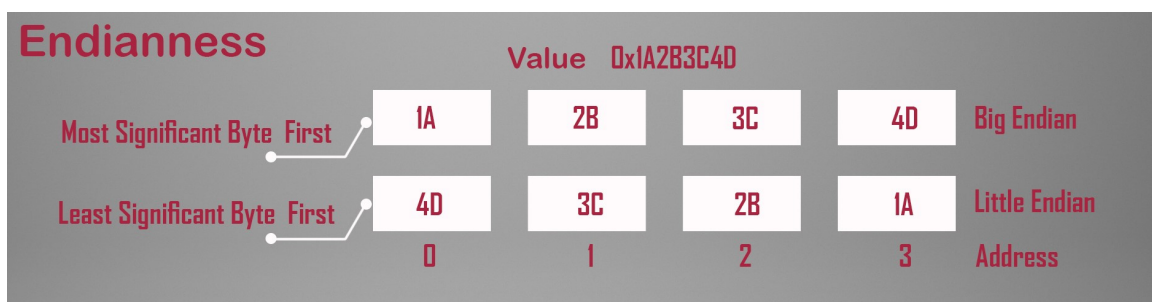
+ .text: chứa các lệnh thực thi của chương trình, vùng chỉ-đọc

+ .data và .bss: cho các biến toàn cục, trong đó .data cho các biến static và đã khởi tạo, .bss dành cho các biến chưa khởi tạo

- Stack và heap được khởi tạo:

+ Stack: cấu trúc dữ liệu theo dạng Last In First Out (LIFO)

+ Heap: cấu trúc dữ liệu theo dạng First In First Out (FIFO)



## 3. Stack Overflows

- Tràn bộ đệm trên stack: Một trong những phương pháp khai thác phần mềm phổ biến và được hiểu nhất

- Mặc dù là lỗ hổng được hiểu nhất và có nhiều tài liệu liên quan nhất, lỗ hổng stack overflow vẫn tồn tại phổ biến trong các phần mềm ngày nay

- Stack overflows – tràn bộ đệm xảy ra do không có cơ chế kiểm soát giới hạn trên buffer trong ngôn ngữ C/C++. Nói cách khác, ngôn ngữ C và các dẫn xuất của nó không có các chức năng sẵn có để đảm bảo dữ liệu đang được sao chép đến buffer sẽ không vượt quá kích thước giới hạn của buffer

=> Hậu quả, nếu người thiết kế chương trình không lập trình chương trình để kiểm tra input vượt quá kích thước, có thể sẽ lấp đầy bộ đệm, và thậm chí nếu đủ lớn, có thể ghi vượt qua giới hạn của buffer

### 3.1. Cách thực hiện khai thác/ Ví dụ

- Bước 1: Tìm offset đến địa chỉ trả về ( thanh ghi eip )

A screenshot of a computer program

Description automatically generated

- Bước 2: Tìm địa chỉ hàm ta cần trả về

A screenshot of a computer screen

Description automatically generated

- Bước 3: Đưa input ta cần vào

A screen shot of a computer

Description automatically generated

# Chương 6: Tràn số nguyên & Chuỗi định dạng

## 1. Interger Overflow

- Trong lập trình máy tính, Integer Oveflow xảy ra khi một phép toán số học cố gắng tạo một giá trị số nằm ngoài phạm vi có thể được biểu diễn bằng một số chữ số nhất định - cao hơn giá trị lớn nhất hoặc thấp hơn giá trị nhỏ nhất có thể biểu diễn

=> Integer Overflow hiểu đơn giản là lỗ hổng trong việc máy tính lưu trữ và tính toán các con số gây ra sai lệch và khiến cho chương trình chạy sai với mong muốn của lập trình viên

### 1.1. Khai thác lỗ hổng này thế nào

- Làm sai lệch các phép tính nhằm thay đổi luồng chạy chương trình

- Lợi dụng để ghi đè lên các vị trí bên ngoài mảng

### 1.2. Cách thực hiện khai thác / Ví dụ

- Cho đoạn code mẫu có lỗ hổng như sau:

A computer screen shot of a program code

Description automatically generated

- Ta sẽ có đoạn code như sau để khai thác lấy flag

A screen shot of a computer program

Description automatically generated

+ Ta sẽ liên tục gửi dữ liệu vào chương trình, vì count được khai báo unsigned char, nên nó sẽ biểu diễn từ 0 đến 255, vậy khi người dùng fake thứ 256 gửi dữ liệu tới thì integer overflow sẽ xảy ra, khiến giá trị của count trở về 0 => thành công lấy được flag

- Lưu ý: với kiểu có dấu thì giá trị sẽ nhảy random chứ khong trở ngược về 0 giống có dấu tùy cấu trúc máy tính

## 2. Format String

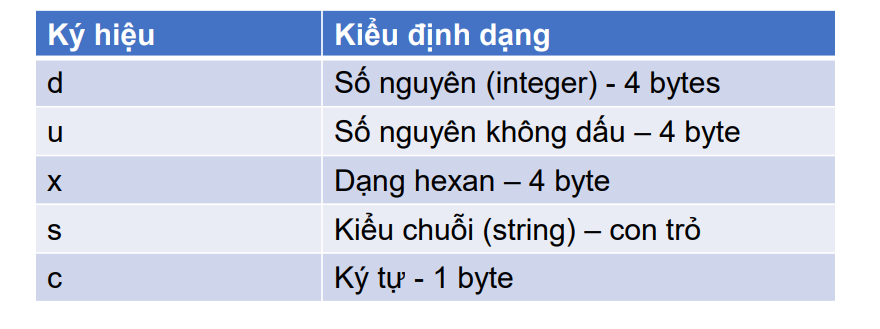
- Format string – Chuỗi định dạng: là một chuỗi ký tự chứa các định dạng đặc biệt, được sử dụng trong lập trình để chèn các giá trị vào chuỗi. Ví dụ, trong C, các định dạng phổ biến như %s, %d, %x cho phép định dạng các kiểu dữ liệu khác nhau như chuỗi, số nguyên, hay số thập lục phân

- Lỗ hổng format string xảy ra khi dữ liệu đầu vào không được kiểm tra cẩn thận và được sử dụng trực tiếp trong các hàm định dạng chuỗi (như printf, fprintf trong C). Điều này cho phép kẻ tấn công kiểm soát chuỗi định dạng, gây ra các hành vi không mong muốn như:

+ Đọc dữ liệu từ bộ nhớ

+ Ghi dữ liệu vào bộ nhớ

+ Thực thi mã độc



### 2.1. Cách khai thác / Ví dụ

- Crash chương trình: %s, đọc dữ liệu trên stack: %x, đọc dữ liệu in tại 1 địa chỉ: %x … %s, ghi dữ liệu: %x … %n

- Bước 1:

# Chương 7: Off-by-one & Return-to-lib-c

## 1. Off-by-one ( Giới hạn kích thước buffer có thể bị ghi đè, không đủ để ghi đè return addr )

- Tấn công tràn bộ đệm trên stack đôi khi không cho phép ghi đè trực tiếp địa chỉ trả về.

- Ví dụ:

A screenshot of a computer code

Description automatically generated

+ Dùng <= thay vì < → có thể nhập kí tự thứ 257, nhưng chưa đủ để ghi đè được return addr -> Cần tấn công kiểu gián tiếp

- Nếu byte thấp nhất trong saved ebp của bar thay đổi thành giá trị nhỏ hơn, ví dụ 0x00? -> Saved ebp sẽ trỏ đến 1 vị trí nào đó thấp hơn stack frame gốc của foo

### 1.1. Ý tưởng

- Ghi 1 stack frame giả vào bộ nhớ đệm để giả mạo stack frame của foo

- Điều khiển được các biến cục bộ của foo xem như cơ bản điều khiển được foo

- Ghi đè byte cuối của saved ebp của bar để trỏ về stack frame giả

- Thậm chí tốt hơn nếu attacker có 1 return addr có chủ đích bên cạnh stack frame giả

+ Có thể điều hướng sau khi thực thi xong foo

- Có thể khai thác như thế nào?

+ Inject code

+ Gán return address trong stack frame giả của foo thành địa chỉ đoạn code muốn thực thi

### 1.2. Kết luận

- Nếu buffer nằm gần saved ebp, tấn công Off-by-one có thể cho phép ghi đè saved ebp

- Kẻ tấn công sau đó có thể thay đổi saved ebp để trỏ đến 1 stack frame giả

- Với stack frame giả, kẻ tấn công có thể kiểm soát return addr, từ đó kiểm soát luồng thực thi

- Một số trường hợp truy xuất con trỏ không chính xác có thể cho phép ghi đè trực tiếp hoặc gián tiếp các cấu trúc/dữ liệu quan trọng, bao gồm return addr

- Các dạng tấn công vào các lỗ hổng này rất khó ngăn chặn

### 1.3. Ví dụ

- Ví dụ đoạn mã code:

A screenshot of a computer program

Description automatically generated

- Với payload là: AAAAAAAAAAAAAAAA\x01 thì byte cuối là x01 sẽ đè vào is\_admin

## 2. Return-to-lib-c

- Ngữ cảnh: việc truyền code thực thi (shellcode) đôi khi khó khăn trong khai thác lỗ hổng buffer overflow

+ Shellcode quá dài không để được trong buffer

+ Shellcode quá phức tạp

+ Trên stack không cho phép thực thi code

* Tận dụng code có sẵn trong chương trình, hay cụ thể là trong các thư viện C (libc)

- Mục tiêu: Ghi đè return address để điều hướng luồng chương trinh đến các đoạn code có sẵn trong bộ nhớ

- Cho đoạn code mẫu:

A white background with black text

Description automatically generated

+ Vấn đề: Dùng strcpy là 1 hàm sao chép chuỗi nhưng không có cơ chế kiểm tra sự phù hợp độ dài giữa nguồn (argv[1]) và đích (buf)

+ Ý tưởng: mở shell bằng cách gọi hàm system(“/bin/bash”), trong đó system() là 1 hàm libc còn tham số của hàm là command muốn thực thi

+ Cách làm: Đổi return address thành địa chỉ của hàm system, gán tham số đầu tiên thanh địa chỉ chuỗi “/bin/bash”

- Để thực thi chức năng phức tạp, có thể cần gọi nhiều hàm libc => Cần “tạo” stack frame cho mỗi hàm được gọi và sử dụng return addr để nối thành chuỗi gọi hàm

### 2.1. Cách khai thác / Ví dụ

- Đoạn code mẫu:

A screenshot of a computer

Description automatically generated

- Bước 1: Tìm địa chỉ hàm system: p system

- Bước 2: Tìm địa chỉ hàm exit: p exit

- Bước 3: Tìm chuỗi /bin/sh: find "/bin/sh"

- Bước 4: Tìm offset ghi đè return address ( thanh eip )

- Bước 5: Xây dựng payload: payload = padding + system\_addr + exit\_addr + bin\_sh\_addr

+ Cấu trúc câu lệnh sẽ là: padding hết offset, gọi địa chỉ hàm system, gọi exit\_addr => địa chỉ trả về sau khi system thực hiện xong, gọi địa chỉ /bin/sh đưa vào hàm system làm tham số

# Chương 8: ROP & Heap overflow

## 1. ROP ( Return Oriented Programming )

- Đôi khi tấn công tràn bộ đệm trên stack không cho phép thực thi code trên stack

+ DEP – Data Execution Prevention: Một kỹ thuật giảm thiểu tấn công bằng cách đảm bảo chỉ có section chứa code mới được gán nhãn là thực thi được

* Không có section/segment bộ nhớ nào được phép cùng lúc cho phép Ghi và Thực thi: W^X
* Các section dữ liệu (data): stack, heap, .bss, .ro, .data
* Các section chứa code: .text, .plt

### 1.1. Ý tưởng

- Tận dụng code có sẵn trong chương trình

**\* Vì sao không tấn công return-to-libc**

- Cần có hàm libc

- Thực thi các hàm libc có thể có tác dụng phụ

- Address randomization có thể khiến các địa chỉ hàm libc khó đoán

**\* ROP**

- Điểm khác biệt: không cần dùng toàn bộ hàm, chỉ sử dụng các đoạn code “đặc biệt” có các lệnh cần thiết – gadget

+ Có thể xem là các hàm nhỏ

+ Luôn kết thúc bằng lệnh ret

**\* Gadget**

- Là một chuỗi liên tục các lệnh assembly có ý nghĩa

- Kết thúc bằng lệnh ret

- VD:

A screenshot of a computer code

Description automatically generated

- Nhiều gadgets được nối (chain) với nhau để tạo thành 1 chuỗi lệnh hiện thực hành vi của attacker tương tự như shellcode => ROP Chain

\* **Cách tìm Gadget**

- Gadget có thể là các lệnh trong các hàm

- Gadget có thể được giấu bên trong các byte của các lệnh assembly khác

**\* ROP chain hoạt động như thế nào?**

**-** Chain bằng cách sắp xếp các address của chúng ở các vị trí thích hợp trong stack

- Address của gadget 1 sẽ ghi đè lên return address của chương trình

- Address của gadget thực thi sau sẽ là return address của gadget thực thi trước

### 1.2. Cách khai thác / Ví dụ

- Bước 1: tìm gadget cần sử dụng: ROPgadget –binary

- Bước 2: Tìm offset để overwrite địa chỉ trả về

- Bước 3: Tạo payload

A screenshot of a computer program

Description automatically generated

- Bước 4: Gửi payload

## 2. Heap Overflow

- Heap là vùng không gian bộ nhớ được dùng cho thao tác cấp phát động tại thời điểm chạy của chương trình:

+ Cấp phát động vùng nhớ với hàm malloc()

+ Giải phóng vùng nhớ trên heap với hàm free()

- Bộ nhớ để lưu trữ cho các biến con trỏ được cấp phát động bởi các hàm malloc, calloc, realloc

- Kích thước không cố định, có thể tăng giảm do đó đáp ứng được nhu cầu lưu trữ dữ liệu của chương trình

- Dữ liệu sẽ không bị hủy khi hàm thực hiện xong → phải tự tay hủy vùng nhớ (câu lệnh free đối với C, và delete hoặc delete [] đối với C++), nếu không sẽ xảy ra hiện tượng rò rỉ bộ nhớ)

### 2.1. Heap Overflow

- Cơ chế xảy ra lỗi trên Heap giống như cách thức hoạt động trên Stack

- Heap cookies/canaries không có vai trò: Không có địa chỉ “trả về” (return) để bảo vệ

- Trong thực tế, có hàng loạt các thông tin như object/struct nằm ở trên Heap: Bất cứ thứ gì xử lý dữ liệu vừa bị hỏng (corrupted) bây giờ đều có thể trở thành bề mặt tấn công (attack surface) trong ứng dụng

- Người ta thường đặt các con trỏ hàm (function pointer) trong các cấu trúc (struct) thường là con trỏ được cấp phát bằng malloc trên heap: Ghi đè lên một con trỏ hàm trên heap và buộc một codepath gọi chức năng của đối tượng đó

### 2.2. Heap Use-After-Free

- Use-After-Free (UAF): là dạng lỗ hổng mà dữ liệu trên heap đã được giải phóng (free-ed), nhưng có một tham chiếu hay “dangling pointer” được sử dụng

- Thường gặp trong Web browser, các chương trình phức tạp

**% Dangling pointer**

**-** Là một con trỏ còn sót lại trong mã nguồn có tham chiếu tới dữ liệu đã được giải phóng trên Heap; dễ có khả năng được sử dụng lại

- Khi được chỉ định tới địa chỉ tại vùng nhớ đã được giải phóng, không có sự đảm bảo nào về dữ liệu tại đây

- Để khai thác UAF, người ta thường phải cấp phát một kiểu dữ liệu khác của đối tượng lên trên vùng nhớ đã được giải phóng

- Không cần bất kỳ hình thức “phá hỏng bộ nhớ” (memory corruption) nào để khai thác lỗ hổng use-after-free (UAF)

- Nó chỉ đơn giản là một vấn đề hiện thực (implementation issue) trong quản lý sai con trỏ (pointer mismanagement)

- Việc phát hiện UAF trong các chương trình phức tạp là một tác vụ khó, ngay trong môi trường công nghiệp (industry)

+ UAF chỉ tồn tại trên một số trạng thái thực thi của chương trình, do đó việc quét mã nguồn tĩnh để tìm kiếm sẽ khó cho ra kết quả

+ Nó thường hay được tìm thấy thông qua “crash”, tuy nhiên, việc sử dụng Symbolic Execution và Contrainst Solver có thể giúp tìm ra lỗi UAF nhanh hơn

- Heap Spraying là một kỹ thuật làm tăng khả năng khai thác, bằng cách lắp đầy heap với một lượng data chunk lớn tương ứng mã khai thác mà kẻ tấn công dự định đặt xuống bộ nhớ, nó hỗ trợ vượt qua được cơ chế bảo vệ của ASLR

- Heap Spraying tận dụng lợi thế là cho phép đặt shellcode ở một vị trí bất kỳ (có thể dự đoán được) trong bộ nhớ và sau đó “jump” về lại địa chỉ đó để thực thi đoạn shellcode

**Ví dụ:**

**A screenshot of a computer

Description automatically generated**