1. **UAF**

**Use-After-Free (UAF)** là một lỗ hổng liên quan đến việc sử dụng không đúng bộ nhớ động trong quá trình hoạt động của chương trình. Nếu sau khi giải phóng một vùng nhớ, chương trình không xóa con trỏ đến vùng nhớ đó, một kẻ tấn công có thể sử dụng lỗ hổng này để xâm nhập vào chương trình.

* Lỗ hổng UAF xảy ra khi một phần tử từ heap đã được giải phóng, nhưng vẫn được sử dụng sau đó.

**Quy trình thực hiện của lỗ hổng UAF:**

1. Cấp phát bộ nhớ: Chương trình cấp phát một phần của bộ nhớ, thường là từ heap, để lưu trữ dữ liệu nào đó.
2. Sử dụng bộ nhớ: Do bộ nhớ động được cấp phát lại nhiều lần, chương trình cần liên tục kiểm tra xem phần nào của heap là miễn phí và phần nào đã được chiếm dụng. Ở đây, các tiêu đề giúp bằng cách tham chiếu đến các khu vực bộ nhớ đã được cấp phát. Mỗi tiêu đề chứa địa chỉ bắt đầu của khối tương ứng.
3. Giải phóng bộ nhớ: Chương trình giải phóng phần của bộ nhớ, đánh dấu nó là không còn được sử dụng nữa. Lỗ hổng UAF xuất hiện khi chương trình không quản lý đúng các tiêu đề này.
4. Sử dụng sau khi giải phóng: Các con trỏ trong chương trình trỏ đến các bộ dữ liệu trong bộ nhớ động. Nếu một bộ dữ liệu bị xóa hoặc chuyển đến khối khác nhưng con trỏ, thay vì được xóa (đặt thành null), tiếp tục trỏ đến bộ nhớ đã được giải phóng, kết quả là một con trỏ treo. Nếu chương trình sau đó cấp phát cùng một phần nhớ này cho một đối tượng khác (ví dụ, dữ liệu nhập bởi một kẻ tấn công), con trỏ treo này sẽ bây giờ trỏ đến bộ dữ liệu mới này. Nói cách khác, lỗ hổng UAF cho phép thay thế mã.

**Hậu quả** tiềm ẩn của việc khai thác UAF bao gồm:

* Sự cố chương trình.
* Thực thi mã tùy ý.

Kẻ tấn công có thể sử dụng UAF để truyền mã tùy ý - hoặc một tham chiếu đến nó - vào chương trình và điều hướng đến đầu của mã bằng cách sử dụng một con trỏ treo. Như vậy, việc thực thi mã độc hại có thể cho phép attacker kiểm soát hệ thống của nạn nhân.

Việc phát hiện lỗ hổng UAF **khó khăn** do

1. Sự phức tạp vì một đầu vào Proof-of-Concept (PoC) cần kích thích một chuỗi ba sự kiện - cấp phát, giải phóng và sử dụng - trên cùng một vùng nhớ

+ Lan rộng qua nhiều hàm của chương trình được kiểm thử: Lỗ hổng UAF thường không xuất hiện trong một hàm cụ thể mà lan rộng qua nhiều hàm khác nhau. Điều này làm cho việc theo dõi và phân tích trở nên khó khăn hơn, đặc biệt là khi có sự chia sẻ dữ liệu giữa các phần khác nhau của chương trình.

1. sự im lặng (không hiện lỗi segmentation fault).

+ Khó phát hiện: Khi có một lỗ hổng UAF, không nhất thiết phải xảy ra lỗi segmentation fault ngay lập tức. Việc này làm cho việc phát hiện trở nên khó khăn hơn, vì không có tín hiệu rõ ràng về sự cố xảy ra.

1. **FUZZING**

Fuzzing là một kỹ thuật kiểm thử tự động được sử dụng để tìm kiếm lỗ hổng bảo mật, lỗi, hoặc hành vi không mong muốn trong phần mềm bằng cách đưa vào chương trình đầu vào ngẫu nhiên hoặc được tạo tự động. Phương pháp này nhằm mục đích kiểm tra tính ổn định và an toàn của ứng dụng một cách tự động và toàn diện hơn so với các phương pháp kiểm thử thủ công.

Fuzzing có thể được thực hiện ở cấp độ đơn vị (unit fuzzing) hoặc ở cấp độ hệ thống (system fuzzing), tùy thuộc vào mức độ ứng dụng của nó. Các công cụ fuzzing như American Fuzzy Lop (AFL), libFuzzer, honggfuzz, và AFL++ là những công cụ phổ biến được sử dụng trong quá trình thực hiện fuzzing. Fuzzing là một phương pháp hiệu quả để tìm ra các lỗ hổng và cung cấp thông tin quan trọng để cải thiện bảo mật của phần mềm.

1. **AFL**

AFL (American Fuzzy Lop) là một công cụ kiểm thử phần mềm dựa trên kỹ thuật "fuzz testing" hay "fuzzing." Fuzz testing là một phương pháp kiểm thử tự động, nơi đầu vào ngẫu nhiên hoặc được tạo ra tự động được đưa vào một chương trình để phát hiện lỗ hổng bảo mật, lỗi chương trình hoặc lỗi hệ thống.

AFL là một công cụ fuzzing phổ biến được phát triển bởi Michal Zalewski. Nó hoạt động bằng cách tạo ra các đầu vào ngẫu nhiên và theo dõi cách mà chương trình xử lý chúng. Nếu chương trình bị crash hoặc có dấu hiệu của lỗ hổng bảo mật, AFL sẽ tập trung vào việc tạo các đầu vào tương tự để kiểm thử sâu hơn.

So với các công cụ fuzzing có trang bị khác, afl-fuzz được thiết kế để thực tế: nó có hiệu suất thấp, sử dụng nhiều chiến lược fuzzing hiệu quả và các thủ thuật tối ưu hóa nỗ lực, yêu cầu ít hoặc không cần cấu hình, và mượt mà xử lý các tình huống thực tế phức tạp, ví dụ như thư viện phân tích hình ảnh chung hoặc thư viện nén file.