Báo cáo đồ án môn An toàn và an ninh mạng

Lớp học phần: INT3307E 1

Giảng viên: TS. Nguyễn Đại Thọ

Nguyễn Tường Hùng – 23020078

December 21, 2025

Abstract

Your abstract.

I. Introduction

II. Background

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`:

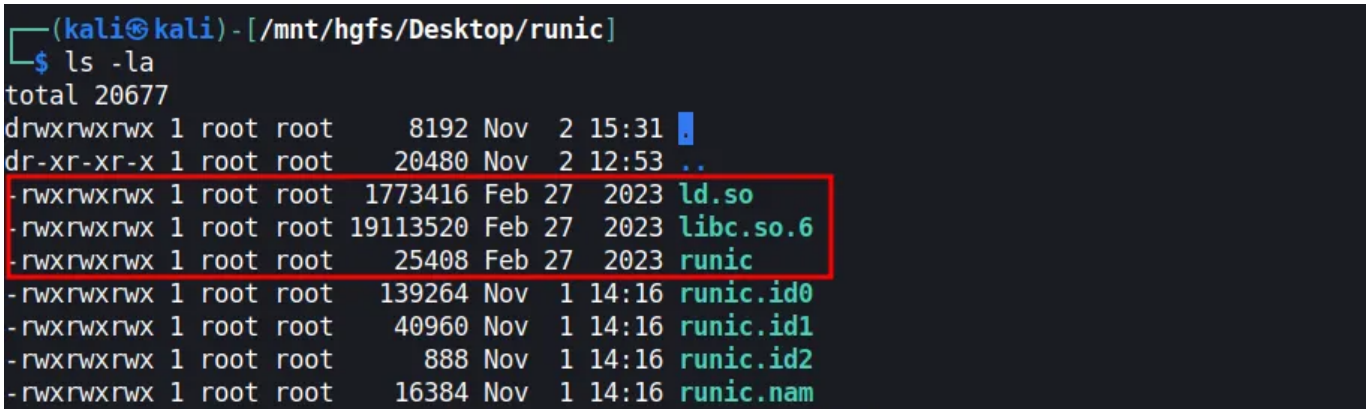

Figure 1: Các file được cung cấp

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.

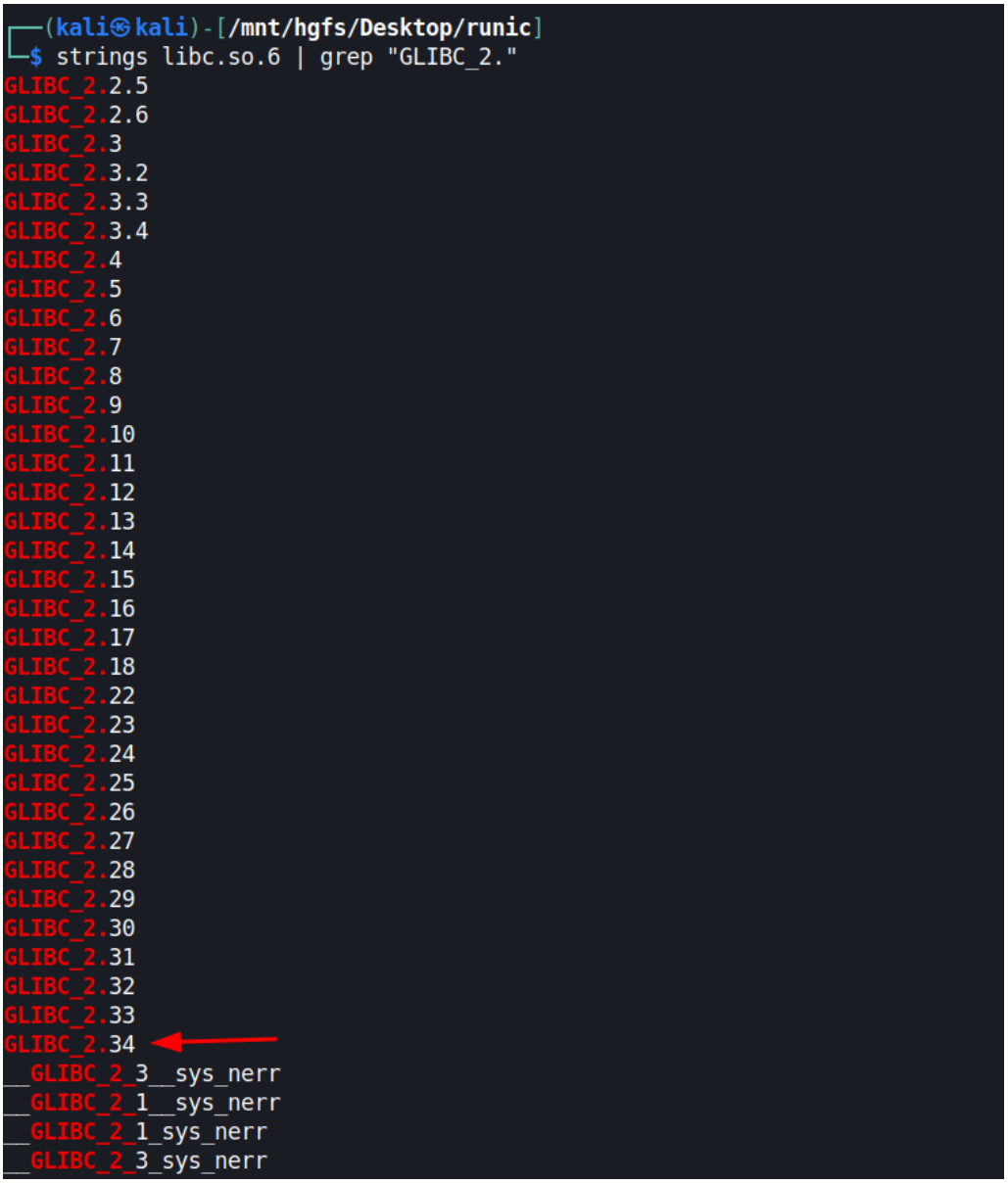

Figure 2: Xác định phiên bản GLIBC

3.2. Binary Mitigations

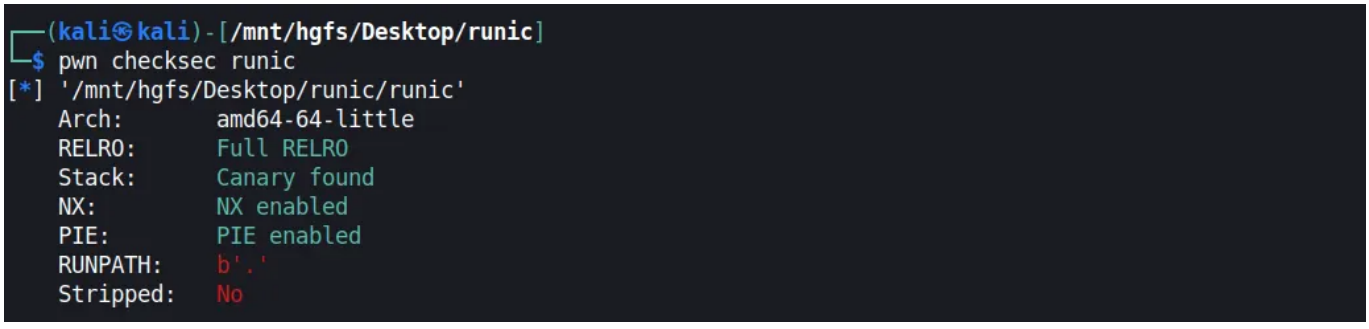

Figure 3: Kết quả checksec

Kết quả 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 sau:

- `Full RELRO` đảm bảo toàn bộ Global Offset Table được đánh dấu read-only sau khi dynamic linker

hoàn tất, ngăn chặn khả năng ghi đè các địa chỉ hàm trong GOT.

- `Canary found` cho biết chương trình sử dụng stack canaries - một giá trị ngẫu nhiên được đặt trên stack để phát hiện stack buffer overflow trước khi hàm return.
- `NX enabled` đánh dấu các vùng nhớ dữ liệu như stack và heap là non-executable, ngăn việc thực thi shellcode trực tiếp tại các vùng này.
- `PIE enabled` cho phép binary được load vào địa chỉ ngẫu nhiên trong memory mỗi lần thực thi, khiến các địa chỉ code và data không thể dự đoán trước.
- `RUNPATH` được đặt là `b'.'` có nghĩa là dynamic linker sẽ tìm shared libraries trong thư mục hiện tại, đảm bảo chương trình load đúng phiên bản libc được cung cấp.
- `Stripped: No` cho biết binary vẫn chứa debug symbols và tên hàm gốc, giúp quá trình phân tích dễ dàng hơn.

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:

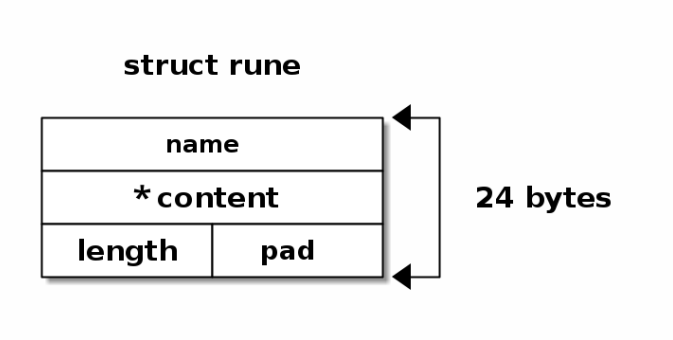

Figure 4: Cấu trúc rune

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.

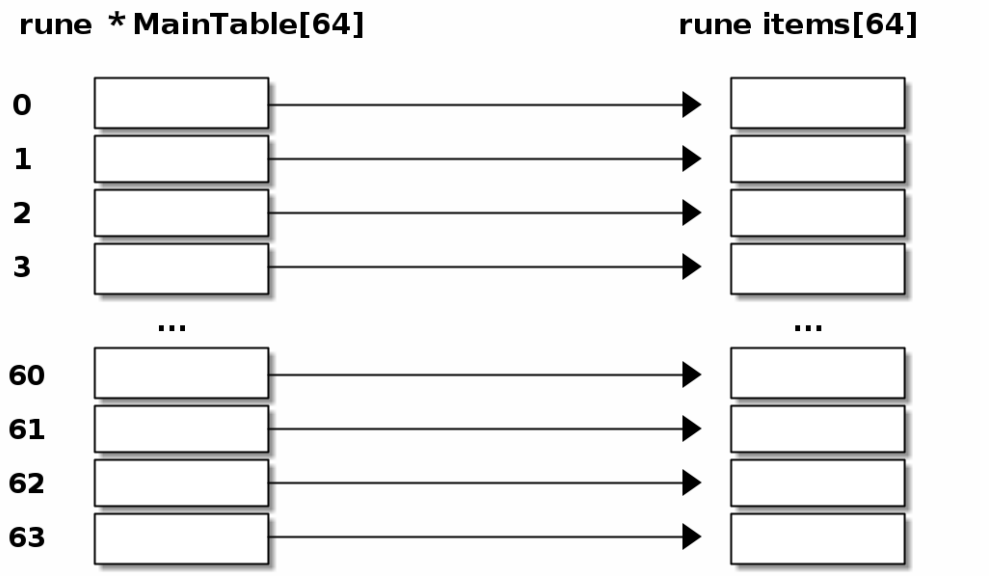

Figure 5: Mối quan hệ giữa mảng con trỏ `MainTable` và mảng dữ liệu `items`

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 với ptmalloc):

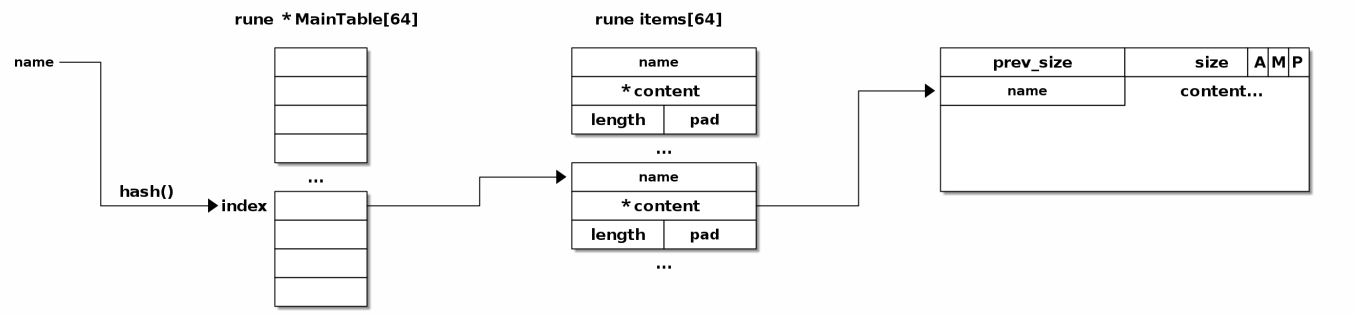

Figure 6: Cái nhìn tổng quát về một rune

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:

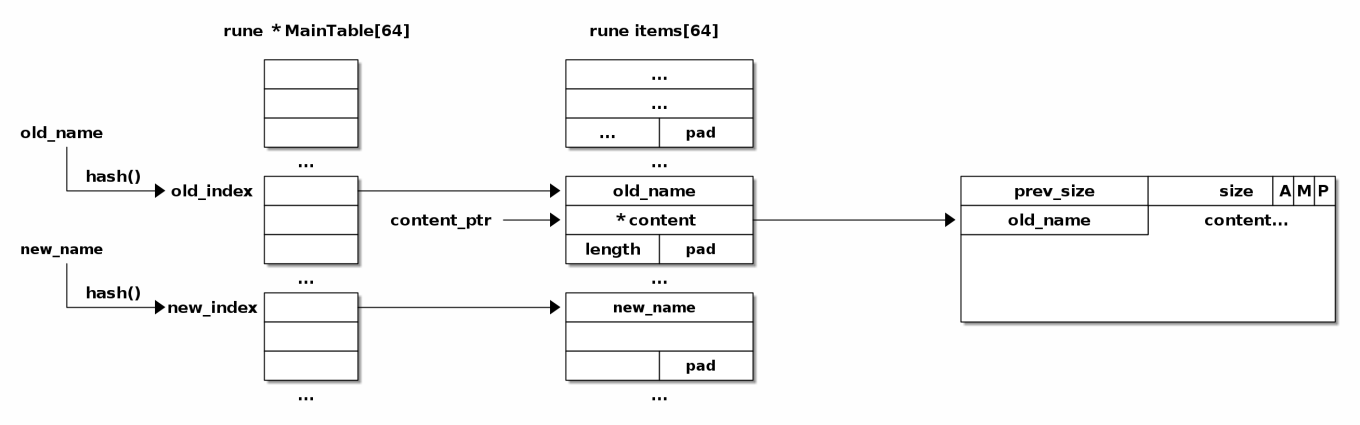

Figure 7: Hình ảnh minh họa 1

```
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:

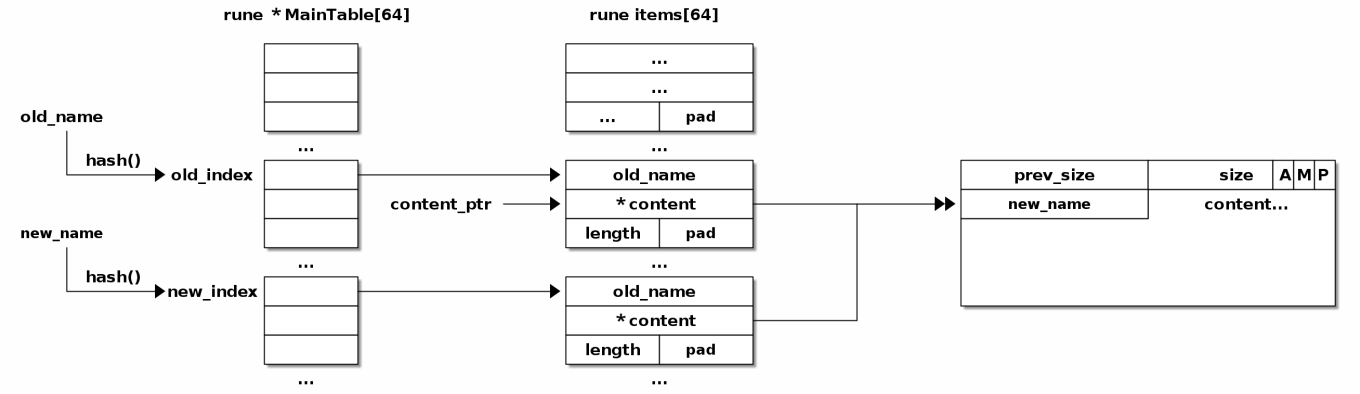

Figure 8: Hình ảnh minh họa 2

```
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):

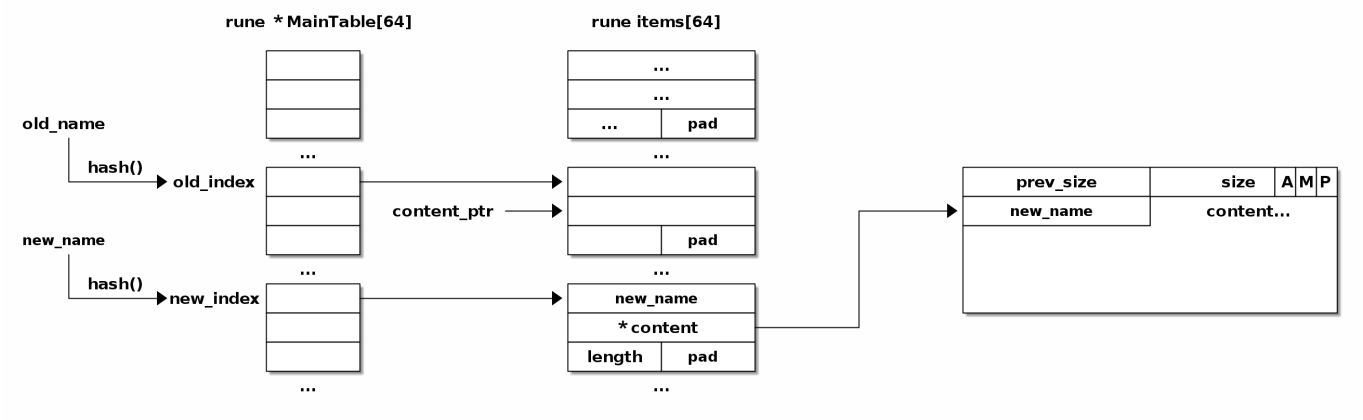

Figure 9: Hình ảnh minh họa 3

```
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`:

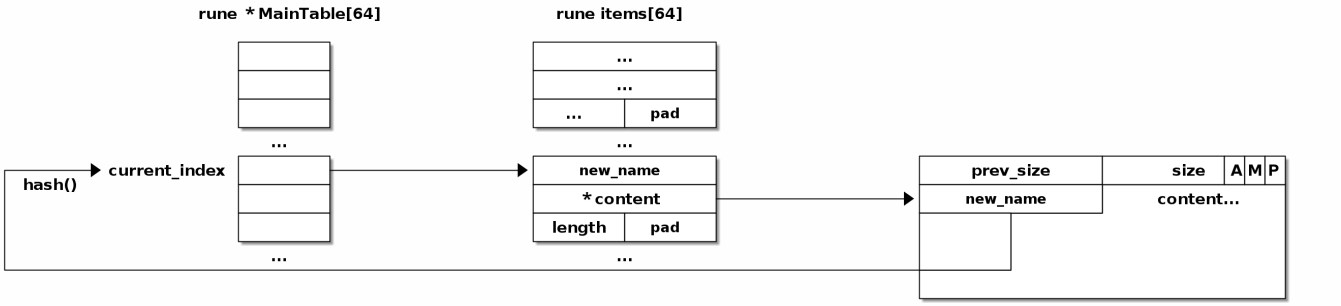

Figure 10: Hình ảnh minh họa 4

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, 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.

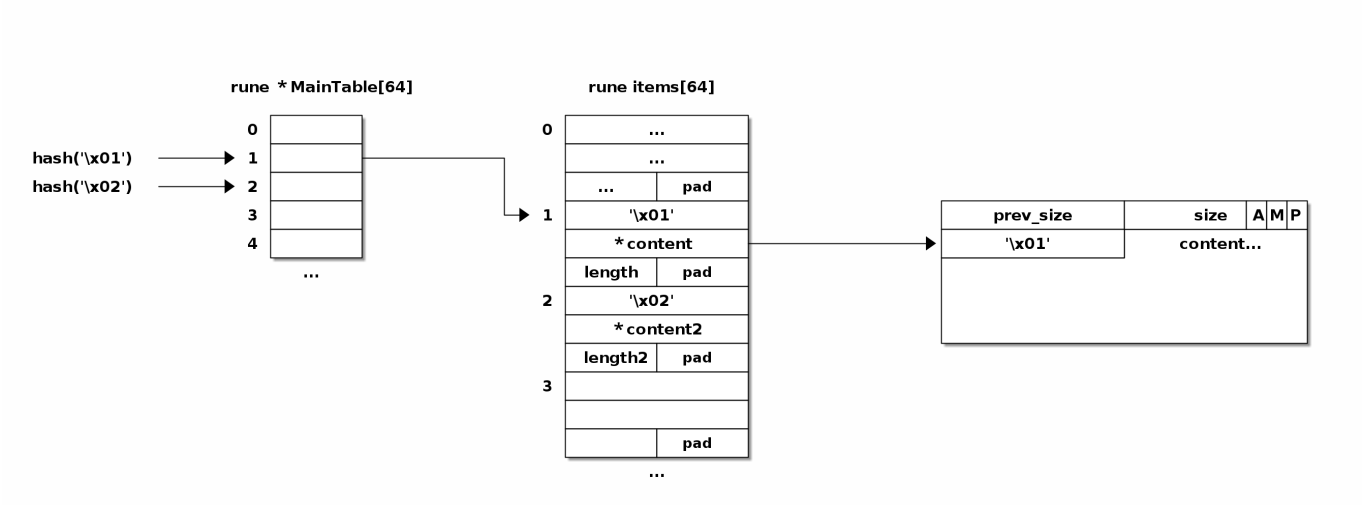

Figure 11: Hình ảnh minh họa 5

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.

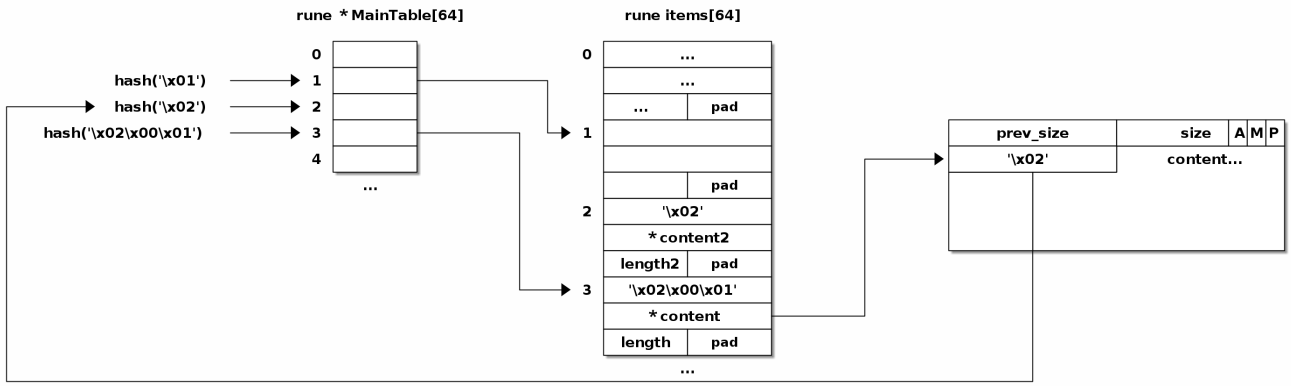

Figure 12: Hình ảnh minh họa 6

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

5.1. Leaking Heap Address

5.2. Leaking Libc Address

5.3. Remote Code Execution

VI. Conclusion

VII. References