

CTF作业：修复FAT32文件系统

Exported on 2025/12/6 at GMT+8 23:44:02 from Cursor (2.1.46)

User

老师布置了个修复fat32文件系统磁盘的作业，给了磁盘恢复工具的同时又给了静谥之眼，优秀的小蓝鲨能发现被破坏的地方以及藏在里面的信息吗？

这是一道CTF的题目 目的是修复损害fat的vmdk文件

Cursor

检查工作目录中的文件：

检查文件大小，并尝试分析该 VMDK 文件：

编写 Python 脚本分析该 VMDK 文件的 FAT32 结构：

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
FAT32文件系统分析工具
用于分析损坏的VMDK文件并查找隐藏信息
"""

import struct
import os

def read_boot_sector(data):
    """读取FAT32引导扇区信息"""
    print("=" * 60)
    print("FAT32 引导扇区分析")
    print("=" * 60)

    # FAT32引导扇区结构
    # 偏移0x00-0x02: Jump指令
    jump = data[0:3]
    print(f"跳转指令: {jump.hex()}")

    # 偏移0x03-0x0A: OEM名称
    oem = data[3:11].decode('ascii', errors='ignore')
    print(f"OEM名称: {oem}")

    # 偏移0x0B-0x0C: 每扇区字节数
    bytes_per_sector = struct.unpack('<H', data[11:13])[0]
    print(f"每扇区字节数: {bytes_per_sector}")

    # 偏移0x0D: 每簇扇区数
    sectors_per_cluster = data[13]
```

```
print(f"每簇扇区数: {sectors_per_cluster}")

# 偏移0x0E-0x0F: 保留扇区数
reserved_sectors = struct.unpack('<H', data[14:16])[0]
print(f"保留扇区数: {reserved_sectors}")

# 偏移0x10: FAT表数量
num_fats = data[16]
print(f"FAT表数量: {num_fats}")

# 偏移0x11-0x12: 根目录条目数 (FAT32中通常为0)
root_entries = struct.unpack('<H', data[17:19])[0]
print(f"根目录条目数: {root_entries}")

# 偏移0x13-0x14: 总扇区数 (小)
total_sectors_small = struct.unpack('<H', data[19:21])[0]
print(f"总扇区数 (小): {total_sectors_small}")

# 偏移0x15: 媒体描述符
media_desc = data[21]
print(f"媒体描述符: 0x{media_desc:02x}")

# 偏移0x16-0x17: 每FAT扇区数 (FAT12/16)
sectors_per_fat_old = struct.unpack('<H', data[22:24])[0]
print(f"每FAT扇区数 (旧): {sectors_per_fat_old}")

# 偏移0x18-0x19: 每磁道扇区数
sectors_per_track = struct.unpack('<H', data[24:26])[0]
print(f"每磁道扇区数: {sectors_per_track}")

# 偏移0x1A-0x1B: 磁头数
num_heads = struct.unpack('<H', data[26:28])[0]
print(f"磁头数: {num_heads}")

# 偏移0x1C-0x1F: 隐藏扇区数
hidden_sectors = struct.unpack('<I', data[28:32])[0]
print(f"隐藏扇区数: {hidden_sectors}")

# 偏移0x20-0x23: 总扇区数 (大)
total_sectors = struct.unpack('<I', data[32:36])[0]
print(f"总扇区数 (大): {total_sectors}")

# 偏移0x24-0x27: 每FAT扇区数 (FAT32)
sectors_per_fat = struct.unpack('<I', data[36:40])[0]
print(f"每FAT扇区数 (FAT32): {sectors_per_fat}")

# 偏移0x28-0x29: 扩展标志
ext_flags = struct.unpack('<H', data[40:42])[0]
print(f"扩展标志: 0x{ext_flags:04x}")

# 偏移0x2A-0x2B: 文件系统版本
fs_version = struct.unpack('<H', data[42:44])[0]
print(f"文件系统版本: {fs_version}")

# 偏移0x2C-0x2F: 根目录簇号
root_cluster = struct.unpack('<I', data[44:48])[0]
```

```

print(f"根目录簇号: {root_cluster}")

# 偏移0x30-0x31: FSInfo扇区号
fsinfo_sector = struct.unpack('<H', data[48:50])[0]
print(f"FSInfo扇区号: {fsinfo_sector}")

# 偏移0x32-0x33: 备份引导扇区
backup_boot = struct.unpack('<H', data[50:52])[0]
print(f"备份引导扇区: {backup_boot}")

# 偏移0x52-0x59: 卷标
volume_label = data[71:82].decode('ascii', errors='ignore').strip()
print(f"卷标: {volume_label}")

# 偏移0x5A-0x61: 文件系统类型
fs_type = data[82:90].decode('ascii', errors='ignore').strip()
print(f"文件系统类型: {fs_type}")

print("=" * 60)

return {
    'bytes_per_sector': bytes_per_sector,
    'sectors_per_cluster': sectors_per_cluster,
    'reserved_sectors': reserved_sectors,
    'num_fats': num_fats,
    'sectors_per_fat': sectors_per_fat,
    'root_cluster': root_cluster,
    'total_sectors': total_sectors if total_sectors > 0 else
total_sectors_small,
    'fsinfo_sector': fsinfo_sector,
    'backup_boot': backup_boot
}

def analyze_fat_table(data, boot_info):
    """分析FAT表"""
    print("\n" + "=" * 60)
    print("FAT表分析")
    print("=" * 60)

    bytes_per_sector = boot_info['bytes_per_sector']
    reserved_sectors = boot_info['reserved_sectors']
    sectors_per_fat = boot_info['sectors_per_fat']

    # FAT表起始位置
    fat_start = reserved_sectors * bytes_per_sector
    fat_size = sectors_per_fat * bytes_per_sector

    print(f"FAT表起始偏移: 0x{fat_start:x} ({fat_start} 字节)")
    print(f"FAT表大小: {fat_size} 字节")

    # 读取FAT表前几个条目
    fat_data = data[fat_start:fat_start + min(fat_size, 1024)]
    print(f"\nFAT表前32个条目 (前128字节):")
    for i in range(0, min(32, len(fat_data) // 4)):
        offset = i * 4
        if offset + 4 <= len(fat_data):

```

```

        entry = struct.unpack('<I', fat_data[offset:offset+4])[0]
        entry_val = entry & 0x0FFFFFFF # FAT32使用低28位
        if entry_val != 0:
            print(f" FAT[{i}]: 0x{entry_val:08X} ({entry_val})")

    return fat_start, fat_size

def search_hidden_data(data):
    """搜索隐藏的数据和异常模式"""
    print("\n" + "=" * 60)
    print("搜索隐藏信息和异常模式")
    print("=" * 60)

    # 搜索常见的CTF标志格式
    patterns = [
        (b'flag{', 'flag格式'),
        (b'FLAG{', 'FLAG格式'),
        (b'ISCTF{', 'ISCTF格式'),
        (b'isctf{', 'isctf格式'),
        (b'ctf{', 'ctf格式'),
        (b'CTF{', 'CTF格式'),
    ]

    for pattern, desc in patterns:
        pos = data.find(pattern)
        if pos != -1:
            # 尝试提取完整标志
            end_pos = data.find(b'}', pos)
            if end_pos != -1:
                flag = data[pos:end_pos+1].decode('ascii', errors='ignore')
                print(f"\n发现{desc}标志在偏移 0x{pos:X} ({pos} 字节):")
                print(f" 内容: {flag}")
                # 显示上下文
                context_start = max(0, pos - 32)
                context_end = min(len(data), end_pos + 32)
                context = data[context_start:context_end]
                print(f" 上下文 (十六进制):")
                print(f" {context.hex()}")

    # 搜索可打印ASCII字符串
    print("\n搜索可打印ASCII字符串 (长度>=10):")
    current_str = b''
    strings_found = []

    for i, byte in enumerate(data):
        if 32 <= byte <= 126: # 可打印ASCII
            current_str += bytes([byte])
        else:
            if len(current_str) >= 10:
                strings_found.append((i - len(current_str), current_str))
            current_str = b''

    # 显示前20个较长的字符串
    strings_found.sort(key=lambda x: len(x[1]), reverse=True)
    for offset, s in strings_found[:20]:
        try:

```

```

        decoded = s.decode('ascii')
        if any(keyword in decoded.lower() for keyword in ['flag', 'secret',
'hidden', 'key', 'password', 'isctf']):
            print(f" 偏移 0x{offset:X}: {decoded[:100]}")
    except:
        pass

def analyze_root_directory(data, boot_info):
    """分析根目录"""
    print("\n" + "=" * 60)
    print("根目录分析")
    print("=" * 60)

    bytes_per_sector = boot_info['bytes_per_sector']
    sectors_per_cluster = boot_info['sectors_per_cluster']
    reserved_sectors = boot_info['reserved_sectors']
    sectors_per_fat = boot_info['sectors_per_fat']
    num_fats = boot_info['num_fats']
    root_cluster = boot_info['root_cluster']

    # 计算数据区起始位置
    data_start = (reserved_sectors + num_fats * sectors_per_fat) *
bytes_per_sector

    # 根目录起始位置（从根目录簇号计算）
    cluster_size = sectors_per_cluster * bytes_per_sector
    root_start = data_start + (root_cluster - 2) * cluster_size

    print(f"数据区起始偏移: 0x{data_start:X} ({data_start} 字节)")
    print(f"根目录起始偏移: 0x{root_start:X} ({root_start} 字节)")
    print(f"簇大小: {cluster_size} 字节")

    # 读取根目录条目（每个条目32字节）
    root_data = data[root_start:root_start + cluster_size]

    print(f"\n根目录条目（前512字节）:")
    for i in range(0, min(16, len(root_data) // 32)):
        offset = i * 32
        entry = root_data[offset:offset+32]

        # 检查是否为空条目
        if entry[0] == 0x00 or entry[0] == 0xE5:
            if entry[0] == 0xE5:
                print(f" 条目 {i}: [已删除]")
            continue

        # 文件名（8字节）+ 扩展名（3字节）
        filename = entry[0:8].decode('ascii', errors='ignore').strip()
        ext = entry[8:11].decode('ascii', errors='ignore').strip()
        fullname = f"{filename}.{ext}" if ext else filename

        # 属性
        attr = entry[11]
        attr_str = []
        if attr & 0x01: attr_str.append('只读')
        if attr & 0x02: attr_str.append('隐藏!')

```

```

if attr & 0x04: attr_str.append('系统')
if attr & 0x08: attr_str.append('卷标')
if attr & 0x10: attr_str.append('目录')
if attr & 0x20: attr_str.append('归档')

# 起始簇号 (高16位在0x14-0x15, 低16位在0x1A-0x1B)
cluster_high = struct.unpack('<H', entry[20:22])[0]
cluster_low = struct.unpack('<H', entry[26:28])[0]
cluster = (cluster_high << 16) | cluster_low

# 文件大小
file_size = struct.unpack('<I', entry[28:32])[0]

if fullname.strip('.'):
    print(f"  条目 {i}: {fullname}")
    print(f"    属性: 0x{attr:02X} ({', '.join(attr_str) if attr_str else
'普通文件'})")
    print(f"    起始簇: {cluster}")
    print(f"    文件大小: {file_size} 字节")
    print(f"    原始数据: {entry.hex()}")

def hex_dump(data, offset, length=512):
    """十六进制转储"""
    print(f"\n偏移 0x{offset:X} 的十六进制转储 ({length} 字节):")
    for i in range(0, min(length, len(data) - offset), 16):
        hex_str = ' '.join(f'{b:02X}' for b in data[offset+i:offset+i+16])
        ascii_str = ''.join(chr(b) if 32 <= b <= 126 else '.' for b in
data[offset+i:offset+i+16])
        print(f" {offset+i:08X}: {hex_str:<48} {ascii_str}")

def main():
    filename = '1.vmdk'

    if not os.path.exists(filename):
        print(f"错误: 文件 {filename} 不存在")
        return

    print(f"正在分析文件: {filename}")
    print(f"文件大小: {os.path.getsize(filename)} 字节\n")

    with open(filename, 'rb') as f:
        data = f.read()

    # 分析引导扇区
    boot_info = read_boot_sector(data)

    # 分析FAT表
    fat_start, fat_size = analyze_fat_table(data, boot_info)

    # 分析根目录
    analyze_root_directory(data, boot_info)

    # 搜索隐藏数据
    search_hidden_data(data)

    # 显示引导扇区的十六进制转储

```

```

hex_dump(data, 0, 512)

# 显示FAT表开头的十六进制转储
hex_dump(data, fat_start, 512)

# 检查备份引导扇区
if boot_info['backup_boot'] > 0:
    backup_offset = boot_info['backup_boot'] * boot_info['bytes_per_sector']
    print(f"\n检查备份引导扇区 (偏移 0x{backup_offset:X}) :")
    hex_dump(data, backup_offset, 512)

# 检查FSInfo扇区
if boot_info['fsinfo_sector'] > 0:
    fsinfo_offset = boot_info['fsinfo_sector'] *
boot_info['bytes_per_sector']
    print(f"\n检查FSInfo扇区 (偏移 0x{fsinfo_offset:X}) :")
    hex_dump(data, fsinfo_offset, 512)

if __name__ == '__main__':
    main()

```

运行脚本分析 VMDK 文件:

User

python3

Cursor

使用 python3 运行脚本:

文件开头是"KDMV", 说明这是 VMDK 容器文件, 不是原始 FAT32 镜像。需要先解析 VMDK 格式, 提取其中的 FAT32 数据。编写脚本解析 VMDK 并定位 FAT32 数据:

```

#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
VMDK文件解析工具
提取VMDK中的FAT32磁盘数据
"""

import struct
import os

def parse_vmdk_header(data):
    """解析VMDK头部"""
    print("=" * 60)
    print("VMDK文件头部分析")
    print("=" * 60)

# VMDK文件通常以"KDMV"开头 (VMware格式)

```

```
if data[0:4] == b'KDMV':
    print("检测到VMware VMDK格式")

    # 读取VMDK头部结构
    # 偏移0x00-0x03: 魔数 "KDMV"
    magic = data[0:4]
    print(f"魔数: {magic}")

    # 偏移0x04-0x07: 版本
    version = struct.unpack('<I', data[4:8])[0]
    print(f"版本: {version}")

    # 偏移0x08-0x0B: 标志
    flags = struct.unpack('<I', data[8:12])[0]
    print(f"标志: 0x{flags:08X}")

    # 偏移0x0C-0x0F: 容量 (扇区数)
    capacity = struct.unpack('<I', data[12:16])[0]
    print(f"容量: {capacity} 扇区 ({capacity * 512} 字节)")

    # 偏移0x10-0x13: 粒度
    grain_size = struct.unpack('<I', data[16:20])[0]
    print(f"粒度: {grain_size} 扇区")

    # 偏移0x14-0x17: 描述符偏移
    desc_offset = struct.unpack('<I', data[20:24])[0]
    print(f"描述符偏移: 0x{desc_offset:X} ({desc_offset} 字节)")

    # 偏移0x18-0x1B: 描述符大小
    desc_size = struct.unpack('<I', data[24:28])[0]
    print(f"描述符大小: {desc_size} 字节")

    # 偏移0x1C-0x1F: 扇区数
    num_sectors = struct.unpack('<I', data[28:32])[0]
    print(f"扇区数: {num_sectors}")

    # 偏移0x20-0x23: 扇区大小
    sector_size = struct.unpack('<I', data[32:36])[0]
    print(f"扇区大小: {sector_size} 字节")

    # 偏移0x24-0x27: 保留
    reserved = struct.unpack('<I', data[36:40])[0]
    print(f"保留: 0x{reserved:08X}")

    # 偏移0x28-0x2B: 父UUID
    parent_uuid = data[40:56]
    print(f"父UUID: {parent_uuid.hex()}")

    # 偏移0x2C-0x3F: UUID
    uuid = data[56:72]
    print(f"UUID: {uuid.hex()}")

    # 查找实际数据起始位置
    # VMDK稀疏格式中, 数据通常在头部之后
    # 查找FAT32引导扇区的特征 (0xEB跳转指令或0xE9)
    print("\n搜索FAT32引导扇区特征...")
```

```

# FAT32引导扇区特征:
# - 偏移0x00: 0xEB 或 0xE9 (跳转指令)
# - 偏移0x03-0x0A: OEM名称 (通常是"MSWIN4.1"或类似)
# - 偏移0x36-0x39: 每FAT扇区数 (FAT32)
# - 偏移0x52-0x59: "FAT32"字符串

for offset in range(0, min(len(data), 65536), 512): # 每512字节检查一次
    if offset + 512 > len(data):
        break

    sector = data[offset:offset+512]

    # 检查跳转指令
    if sector[0] in [0xEB, 0xE9] or (sector[0] == 0xE8 and sector[1] <
0x80):
        # 检查OEM名称
        oem = sector[3:11].decode('ascii', errors='ignore')
        # 检查FAT32标识
        fs_type = sector[82:90].decode('ascii', errors='ignore').strip()

        if 'FAT' in fs_type or 'MSWIN' in oem or 'MSDOS' in oem:
            print(f"\n发现可能的FAT32引导扇区在偏移: 0x{offset:X} ({offset}
字节)")

            print(f"  跳转指令: 0x{sector[0]:02X}")
            print(f"  OEM名称: {oem}")
            print(f"  文件系统类型: {fs_type}")

            # 显示引导扇区内容
            print(f"\n引导扇区十六进制转储:")
            for i in range(0, 512, 16):
                hex_str = ' '.join(f'{b:02X}' for b in sector[i:i+16])
                ascii_str = ''.join(chr(b) if 32 <= b <= 126 else '.' for
b in sector[i:i+16])
                print(f"  {i:04X}: {hex_str:<48} {ascii_str}")

            return offset

print("\n未找到标准的FAT32引导扇区, 尝试其他方法...")

# 如果没找到, 可能VMDK是稀疏格式, 需要查找数据段
# 或者直接尝试从某个固定偏移开始
# 通常VMDK头部后就是数据
possible_offsets = [512, 1024, 2048, 4096, 8192]
for offset in possible_offsets:
    if offset < len(data):
        sector = data[offset:offset+512]
        print(f"\n检查偏移 0x{offset:X}:")
        print(f"  前16字节: {sector[0:16].hex()}")
        if sector[0] in [0xEB, 0xE9, 0xE8]:
            oem = sector[3:11].decode('ascii', errors='ignore')
            print(f"  OEM: {oem}")
            if 'MS' in oem or 'FAT' in sector[82:90].decode('ascii',
errors='ignore'):
                return offset

```

```

        return None

    else:
        print("不是标准的VMDK格式，尝试直接作为FAT32分析...")
        return 0

def extract_fat32_from_vmdk(vmdk_data, fat32_offset):
    """从VMDK中提取FAT32数据"""
    if fat32_offset is None:
        print("无法确定FAT32数据起始位置")
        return None

    print(f"\n从偏移 0x{fat32_offset:x} 提取FAT32数据...")

    # 提取从该偏移开始的所有数据
    fat32_data = vmdk_data[fat32_offset:]

    # 保存为临时文件
    output_file = 'fat32_disk.img'
    with open(output_file, 'wb') as f:
        f.write(fat32_data)

    print(f"FAT32数据已提取到: {output_file}")
    print(f"提取的数据大小: {len(fat32_data)} 字节")

    return fat32_data

def main():
    filename = '1.vmdk'

    if not os.path.exists(filename):
        print(f"错误: 文件 {filename} 不存在")
        return

    print(f"正在解析VMDK文件: {filename}")
    print(f"文件大小: {os.path.getsize(filename)} 字节\n")

    with open(filename, 'rb') as f:
        data = f.read()

    # 解析VMDK头部
    fat32_offset = parse_vmdk_header(data)

    # 提取FAT32数据
    if fat32_offset is not None:
        fat32_data = extract_fat32_from_vmdk(data, fat32_offset)

        if fat32_data:
            print("\n现在可以分析提取的FAT32数据了")
            print("请运行 analyze_fat32.py 并指定提取的文件")

if __name__ == '__main__':
    main()

```

运行脚本解析 VMDK 文件:

在偏移 0x200 处发现 "Disk Descriptor"。检查该区域并查找实际数据部分:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
深度分析VMDK文件, 查找FAT32数据
"""

import struct
import os
import re

def hex_dump(data, offset, length=512):
    """十六进制转储"""
    print(f"\n偏移 0x{offset:X} 的十六进制转储 ({length} 字节):")
    for i in range(0, min(length, len(data) - offset), 16):
        hex_str = ' '.join(f'{b:02X}' for b in data[offset+i:offset+i+16])
        ascii_str = ''.join(chr(b) if 32 <= b <= 126 else '.' for b in
data[offset+i:offset+i+16])
        print(f" {offset+i:08X}: {hex_str:<48} {ascii_str}")

def find_all_strings(data, min_len=4):
    """查找所有可打印字符串"""
    strings = []
    current = b''
    start_pos = 0

    for i, byte in enumerate(data):
        if 32 <= byte <= 126:
            if len(current) == 0:
                start_pos = i
                current += bytes([byte])
            else:
                if len(current) >= min_len:
                    try:
                        s = current.decode('ascii')
                        strings.append((start_pos, s))
                    except:
                        pass
                current = b''

    if len(current) >= min_len:
        try:
            s = current.decode('ascii')
            strings.append((start_pos, s))
        except:
            pass

    return strings

def search_fat32_signatures(data):
    """搜索FAT32特征"""
```

```

print("=" * 60)
print("搜索FAT32特征")
print("=" * 60)

# FAT32引导扇区特征
signatures = [
    (b'MSWIN4.1', 'MSWIN4.1 OEM'),
    (b'MSDOS5.0', 'MSDOS5.0 OEM'),
    (b'FAT32', 'FAT32标识'),
    (b'FAT16', 'FAT16标识'),
    (b'FAT12', 'FAT12标识'),
]

for sig, desc in signatures:
    pos = data.find(sig)
    if pos != -1:
        # 检查是否在扇区边界（512字节对齐）
        sector_offset = pos % 512
        sector_start = pos - sector_offset

        print(f"\n发现{desc}在偏移 0x{pos:X}")
        print(f" 所在扇区起始: 0x{sector_start:X}")
        print(f" 扇区内偏移: {sector_offset}")

        # 检查是否是引导扇区
        if sector_offset == 3 and sector_start + 512 <= len(data):
            sector = data[sector_start:sector_start+512]
            if sector[0] in [0xEB, 0xE9, 0xE8]:
                print(f"  ✓ 可能是引导扇区（跳转指令: 0x{sector[0]:02X} ")
                hex_dump(data, sector_start, 512)
                return sector_start

    return None

def analyze_vmdk_structure(data):
    """分析VMDK文件结构"""
    print("=" * 60)
    print("VMDK文件结构分析")
    print("=" * 60)

    # 显示文件开头
    hex_dump(data, 0, 1024)

    # 查找描述符部分
    desc_pos = data.find(b'# Disk Descriptor')
    if desc_pos != -1:
        print(f"\n发现磁盘描述符在偏移: 0x{desc_pos:X}")
        hex_dump(data, desc_pos, 512)

        # 查找描述符结束（通常是空行或特定标记）
        desc_end = data.find(b'\n\n', desc_pos)
        if desc_end == -1:
            desc_end = data.find(b'# The Disk Data Base', desc_pos)
        if desc_end != -1:
            print(f"\n描述符部分结束于: 0x{desc_end:X}")
            hex_dump(data, desc_end, 512)

```

```

# 查找所有可打印字符串
print("\n查找所有可打印字符串（长度>=8）：")
strings = find_all_strings(data, 8)
for pos, s in strings[:30]: # 显示前30个
    if any(keyword in s.lower() for keyword in ['fat', 'disk', 'sector',
'cluster', 'boot', 'flag', 'secret']):
        print(f" 偏移 0x{pos:X}: {s[:80]}")

# 搜索FAT32特征
fat32_offset = search_fat32_signatures(data)

if fat32_offset:
    return fat32_offset

# 如果没找到，尝试查找数据段
# VMDK稀疏格式中，数据通常在描述符之后
print("\n尝试查找数据段...")

# 查找可能的扇区边界（512字节对齐的非零数据）
for offset in range(0, min(len(data), 100000), 512):
    if offset + 512 > len(data):
        break

    sector = data[offset:offset+512]

    # 检查是否有引导扇区特征
    if sector[0] in [0xEB, 0xE9, 0xE8]:
        # 检查扇区末尾是否有0x55AA（引导扇区签名）
        if len(sector) >= 512 and struct.unpack('<H', sector[510:512])[0] ==
0xAA55:
            print(f"\n发现可能的引导扇区在偏移：0x{offset:X}")
            hex_dump(data, offset, 512)

            # 检查OEM和文件系统类型
            oem = sector[3:11].decode('ascii', errors='ignore')
            fs_type = sector[82:90].decode('ascii', errors='ignore').strip()
            print(f" OEM: {oem}")
            print(f" 文件系统类型: {fs_type}")

            if 'FAT' in fs_type or 'MS' in oem:
                return offset

    return None

def main():
    filename = '1.vmdk'

    if not os.path.exists(filename):
        print(f"错误：文件 {filename} 不存在")
        return

    print(f"正在深度分析VMDK文件：{filename}")
    print(f"文件大小：{os.path.getsize(filename)} 字节\n")

    with open(filename, 'rb') as f:

```

```

data = f.read()

# 分析VMDK结构
fat32_offset = analyze_vmdk_structure(data)

if fat32_offset:
    print(f"\n{'='*60}")
    print(f"找到FAT32数据起始位置: 0x{fat32_offset:X}")
    print(f"{'='*60}")

    # 提取FAT32数据
    fat32_data = data[fat32_offset:]
    output_file = 'fat32_disk.img'
    with open(output_file, 'wb') as f:
        f.write(fat32_data)
    print(f"FAT32数据已提取到: {output_file}")
    print(f"提取的数据大小: {len(fat32_data)} 字节")
else:
    print("\n未找到FAT32数据, 可能需要进一步分析")

if __name__ == '__main__':
    main()

```

已找到 FAT32 数据起始位置 (偏移 0x60000) 。分析提取的 FAT32 镜像, 查找损坏和隐藏信息:

```

#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
分析提取的FAT32磁盘镜像
查找损坏和隐藏信息
"""

import struct
import os

def read_boot_sector(data):
    """读取FAT32引导扇区信息"""
    print("=" * 60)
    print("FAT32 引导扇区详细分析")
    print("=" * 60)

    if len(data) < 512:
        print("错误: 数据太小, 不是有效的引导扇区")
        return None

    # 检查引导扇区签名
    boot_signature = struct.unpack('<H', data[510:512])[0]
    if boot_signature != 0xAA55:
        print(f"警告: 引导扇区签名不正确: 0x{boot_signature:04X} (应该是0xAA55)")
    else:
        print("✓ 引导扇区签名正确: 0xAA55")

    # 跳转指令
    jump = data[0:3]

```

```
print(f"跳转指令: {jump.hex()} ({jump[0]:02X}")

# OEM名称
oem = data[3:11].decode('ascii', errors='ignore')
print(f"OEM名称: {oem}")

# 每扇区字节数
bytes_per_sector = struct.unpack('<H', data[11:13])[0]
print(f"每扇区字节数: {bytes_per_sector}")

# 每簇扇区数
sectors_per_cluster = data[13]
print(f"每簇扇区数: {sectors_per_cluster}")

# 保留扇区数
reserved_sectors = struct.unpack('<H', data[14:16])[0]
print(f"保留扇区数: {reserved_sectors}")

# FAT表数量
num_fats = data[16]
print(f"FAT表数量: {num_fats}")

# 根目录条目数 (FAT32中通常为0)
root_entries = struct.unpack('<H', data[17:19])[0]
print(f"根目录条目数: {root_entries}")

# 总扇区数 (小)
total_sectors_small = struct.unpack('<H', data[19:21])[0]
print(f"总扇区数 (小): {total_sectors_small}")

# 媒体描述符
media_desc = data[21]
print(f"媒体描述符: 0x{media_desc:02X}")

# 每FAT扇区数 (FAT12/16)
sectors_per_fat_old = struct.unpack('<H', data[22:24])[0]
print(f"每FAT扇区数 (旧): {sectors_per_fat_old}")

# 每磁道扇区数
sectors_per_track = struct.unpack('<H', data[24:26])[0]
print(f"每磁道扇区数: {sectors_per_track}")

# 磁头数
num_heads = struct.unpack('<H', data[26:28])[0]
print(f"磁头数: {num_heads}")

# 隐藏扇区数
hidden_sectors = struct.unpack('<I', data[28:32])[0]
print(f"隐藏扇区数: {hidden_sectors}")

# 总扇区数 (大)
total_sectors = struct.unpack('<I', data[32:36])[0]
print(f"总扇区数 (大): {total_sectors}")

# 每FAT扇区数 (FAT32)
sectors_per_fat = struct.unpack('<I', data[36:40])[0]
```

```

print(f"每FAT扇区数 (FAT32) : {sectors_per_fat}")

# 扩展标志
ext_flags = struct.unpack('<H', data[40:42])[0]
print(f"扩展标志: 0x{ext_flags:04X}")

# 文件系统版本
fs_version = struct.unpack('<H', data[42:44])[0]
print(f"文件系统版本: {fs_version}")

# 根目录簇号
root_cluster = struct.unpack('<I', data[44:48])[0]
print(f"根目录簇号: {root_cluster}")

# FSInfo扇区号
fsinfo_sector = struct.unpack('<H', data[48:50])[0]
print(f"FSInfo扇区号: {fsinfo_sector}")

# 备份引导扇区
backup_boot = struct.unpack('<H', data[50:52])[0]
print(f"备份引导扇区: {backup_boot}")

# 卷序列号
volume_serial = struct.unpack('<I', data[67:71])[0]
print(f"卷序列号: 0x{volume_serial:08X}")

# 卷标
volume_label = data[71:82].decode('ascii', errors='ignore').strip()
print(f"卷标: '{volume_label}'")

# 文件系统类型
fs_type = data[82:90].decode('ascii', errors='ignore').strip()
print(f"文件系统类型: '{fs_type}'")

print("=" * 60)

return {
    'bytes_per_sector': bytes_per_sector,
    'sectors_per_cluster': sectors_per_cluster,
    'reserved_sectors': reserved_sectors,
    'num_fats': num_fats,
    'sectors_per_fat': sectors_per_fat,
    'root_cluster': root_cluster,
    'total_sectors': total_sectors if total_sectors > 0 else
total_sectors_small,
    'fsinfo_sector': fsinfo_sector,
    'backup_boot': backup_boot,
    'volume_serial': volume_serial,
    'volume_label': volume_label
}

def analyze_fat_table(data, boot_info):
    """分析FAT表"""
    print("\n" + "=" * 60)
    print("FAT表分析")
    print("=" * 60)

```

```

bytes_per_sector = boot_info['bytes_per_sector']
reserved_sectors = boot_info['reserved_sectors']
sectors_per_fat = boot_info['sectors_per_fat']
num_fats = boot_info['num_fats']

# FAT表起始位置
fat_start = reserved_sectors * bytes_per_sector
fat_size = sectors_per_fat * bytes_per_sector

print(f"FAT表起始偏移: 0x{fat_start:X} ({fat_start} 字节)")
print(f"每个FAT表大小: {fat_size} 字节")
print(f"FAT表数量: {num_fats}")

if fat_start + fat_size * num_fats > len(data):
    print(f"警告: FAT表超出数据范围!")
    return None, None

# 读取第一个FAT表
fat1_data = data[fat_start:fat_start + fat_size]

print(f"\n第一个FAT表前64个条目:")
for i in range(0, min(64, len(fat1_data) // 4)):
    offset = i * 4
    if offset + 4 <= len(fat1_data):
        entry = struct.unpack('<I', fat1_data[offset:offset+4])[0]
        entry_val = entry & 0x0FFFFFFF # FAT32使用低28位
        status = ""
        if i == 0:
            status = " (媒体描述符)"
        elif entry_val == 0:
            status = " (空闲)"
        elif entry_val == 0x0FFFFFF7:
            status = " (坏簇)"
        elif entry_val >= 0x0FFFFFF8:
            status = " (文件结束)"
        elif entry_val >= 2:
            status = f" (指向簇 {entry_val})"

        if entry_val != 0 or i < 10:
            print(f" FAT[{i}]: 0x{entry_val:08X} ({entry_val}){status}")

# 检查FAT表是否有损坏
print(f"\n检查FAT表完整性...")
zero_entries = sum(1 for i in range(0, len(fat1_data), 4)
                    if struct.unpack('<I', fat1_data[i:i+4])[0] & 0x0FFFFFFF
== 0)
print(f" 空闲条目数: {zero_entries}")

# 如果有第二个FAT表, 比较它们
if num_fats > 1:
    fat2_start = fat_start + fat_size
    fat2_data = data[fat2_start:fat2_start + fat_size]

    if fat1_data == fat2_data:
        print(f"  ✓ 两个FAT表一致")

```

```

else:
    print(f" X 两个FAT表不一致! 可能有损坏或隐藏数据")
    # 找出差异
    diff_count = 0
    for i in range(0, min(len(fat1_data), len(fat2_data)), 4):
        if fat1_data[i:i+4] != fat2_data[i:i+4]:
            diff_count += 1
            if diff_count <= 10:
                e1 = struct.unpack('<I', fat1_data[i:i+4])[0] &
0x0FFFFFFF
                e2 = struct.unpack('<I', fat2_data[i:i+4])[0] &
0x0FFFFFFF
                print(f" 条目 {i//4}: FAT1=0x{e1:08X},
FAT2=0x{e2:08X}")
            print(f" 总差异数: {diff_count}")

    return fat_start, fat_size

def analyze_root_directory(data, boot_info):
    """分析根目录"""
    print("\n" + "=" * 60)
    print("根目录分析")
    print("=" * 60)

    bytes_per_sector = boot_info['bytes_per_sector']
    sectors_per_cluster = boot_info['sectors_per_cluster']
    reserved_sectors = boot_info['reserved_sectors']
    sectors_per_fat = boot_info['sectors_per_fat']
    num_fats = boot_info['num_fats']
    root_cluster = boot_info['root_cluster']

    # 计算数据区起始位置
    data_start = (reserved_sectors + num_fats * sectors_per_fat) *
bytes_per_sector

    # 根目录起始位置 (从根目录簇号计算)
    cluster_size = sectors_per_cluster * bytes_per_sector
    root_start = data_start + (root_cluster - 2) * cluster_size

    print(f"数据区起始偏移: 0x{data_start:X} ({data_start} 字节)")
    print(f"根目录起始偏移: 0x{root_start:X} ({root_start} 字节)")
    print(f"簇大小: {cluster_size} 字节")
    print(f"根目录簇号: {root_cluster}")

    if root_start >= len(data):
        print("警告: 根目录偏移超出数据范围!")
        return

    # 读取根目录条目 (每个条目32字节)
    root_data = data[root_start:root_start + min(cluster_size, len(data) -
root_start)]

    print(f"\n根目录条目:")
    entry_count = 0
    for i in range(0, len(root_data) // 32):
        offset = i * 32

```

```

entry = root_data[offset:offset+32]

# 检查是否为空条目
if entry[0] == 0x00:
    break
if entry[0] == 0xE5:
    print(f" 条目 {i}: [已删除]")
    continue

entry_count += 1

# 文件名 (8字节) + 扩展名 (3字节)
filename = entry[0:8].decode('ascii', errors='ignore').strip()
ext = entry[8:11].decode('ascii', errors='ignore').strip()
fullname = f"{filename}.{ext}" if ext else filename

# 属性
attr = entry[11]
attr_str = []
if attr & 0x01: attr_str.append('只读')
if attr & 0x02: attr_str.append('隐藏')
if attr & 0x04: attr_str.append('系统')
if attr & 0x08: attr_str.append('卷标')
if attr & 0x10: attr_str.append('目录')
if attr & 0x20: attr_str.append('归档')

# 起始簇号 (高16位在0x14-0x15, 低16位在0x1A-0x1B)
cluster_high = struct.unpack('<H', entry[20:22])[0]
cluster_low = struct.unpack('<H', entry[26:28])[0]
cluster = (cluster_high << 16) | cluster_low

# 文件大小
file_size = struct.unpack('<I', entry[28:32])[0]

print(f" 条目 {i}: {fullname}")
print(f" 属性: 0x{attr:02X} ({', '.join(attr_str) if attr_str else '普通文件'})")
print(f" 起始簇: {cluster}")
print(f" 文件大小: {file_size} 字节")
print(f" 原始数据: {entry.hex()}")

print(f"\n总有效条目数: {entry_count}")

def search_hidden_data(data):
    """搜索隐藏的数据和异常模式"""
    print("\n" + "=" * 60)
    print("搜索隐藏信息和异常模式")
    print("=" * 60)

# 搜索常见的CTF标志格式
patterns = [
    (b'flag{', 'flag格式'),
    (b'FLAG{', 'FLAG格式'),
    (b'ISCTF{', 'ISCTF格式'),
    (b'isctf{', 'isctf格式'),
    (b'ctf{', 'ctf格式'),

```

```

    (b'CTF{', 'CTF格式'),
    (b'静谧', '静谧'),
    (b'\xe9\x9d\x99', '静谧(UTF-8)'),
]

found_flags = []
for pattern, desc in patterns:
    pos = 0
    while True:
        pos = data.find(pattern, pos)
        if pos == -1:
            break
        # 尝试提取完整标志
        if pattern[0] in [b'{'[0], b'('[0]]:
            end_pos = data.find(b'}', pos) if b'}' in pattern else
data.find(b')', pos)
        else:
            end_pos = pos + len(pattern) + 50 # 尝试读取更多

        if end_pos != -1 and end_pos < len(data):
            try:
                flag = data[pos:end_pos+1].decode('ascii', errors='ignore')
                found_flags.append((pos, flag, desc))
            except:
                pass
        pos += 1

if found_flags:
    print("\n发现可能的标志:")
    for pos, flag, desc in found_flags:
        print(f"  偏移 0x{pos:X}: {flag[:100]} ({desc})")
else:
    print("未发现标准格式的标志")

# 搜索可打印ASCII字符串
print("\n搜索可打印ASCII字符串（长度>=10，包含关键词）:")
current_str = b''
strings_found = []

for i, byte in enumerate(data):
    if 32 <= byte <= 126: # 可打印ASCII
        current_str += bytes([byte])
    else:
        if len(current_str) >= 10:
            try:
                decoded = current_str.decode('ascii')
                if any(keyword in decoded.lower() for keyword in ['flag',
'secret', 'hidden', 'key', 'password', 'isctf', 'ctf', '静谧', 'quiet']):
                    strings_found.append((i - len(current_str), decoded))
            except:
                pass
        current_str = b''

# 显示找到的字符串
for offset, s in strings_found[:20]:
    print(f"  偏移 0x{offset:X}: {s[:100]}")

```

```

def hex_dump(data, offset, length=512):
    """十六进制转储"""
    print(f"\n偏移 0x{offset:X} 的十六进制转储 ({length} 字节):")
    for i in range(0, min(length, len(data) - offset), 16):
        hex_str = ' '.join(f'{b:02X}' for b in data[offset+i:offset+i+16])
        ascii_str = ''.join(chr(b) if 32 <= b <= 126 else '.' for b in
data[offset+i:offset+i+16])
        print(f" {offset+i:08X}: {hex_str:<48} {ascii_str}")

def main():
    filename = 'fat32_disk.img'

    if not os.path.exists(filename):
        print(f"错误: 文件 {filename} 不存在")
        print("请先运行 deep_analyze_vmdk.py 提取FAT32数据")
        return

    print(f"正在分析FAT32磁盘镜像: {filename}")
    print(f"文件大小: {os.path.getsize(filename)} 字节\n")

    with open(filename, 'rb') as f:
        data = f.read()

    # 分析引导扇区
    boot_info = read_boot_sector(data)

    if boot_info is None:
        return

    # 分析FAT表
    fat_start, fat_size = analyze_fat_table(data, boot_info)

    # 分析根目录
    analyze_root_directory(data, boot_info)

    # 搜索隐藏数据
    search_hidden_data(data)

    # 显示引导扇区的十六进制转储
    hex_dump(data, 0, 512)

    # 检查备份引导扇区
    if boot_info['backup_boot'] > 0:
        backup_offset = boot_info['backup_boot'] * boot_info['bytes_per_sector']
        if backup_offset < len(data):
            print(f"\n检查备份引导扇区 (偏移 0x{backup_offset:X}):")
            hex_dump(data, backup_offset, 512)

    # 检查FSInfo扇区
    if boot_info['fsinfo_sector'] > 0:
        fsinfo_offset = boot_info['fsinfo_sector'] *
boot_info['bytes_per_sector']
        if fsinfo_offset < len(data):
            print(f"\n检查FSInfo扇区 (偏移 0x{fsinfo_offset:X}):")
            hex_dump(data, fsinfo_offset, 512)

```

```
if __name__ == '__main__':
    main()
```

修复语法错误:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
分析提取的FAT32磁盘镜像
查找损坏和隐藏信息
"""

import struct
import os

def read_boot_sector(data):
    """读取FAT32引导扇区信息"""
    print("=" * 60)
    print("FAT32 引导扇区详细分析")
    print("=" * 60)

    if len(data) < 512:
        print("错误: 数据太小, 不是有效的引导扇区")
        return None

    # 检查引导扇区签名
    boot_signature = struct.unpack('<H', data[510:512])[0]
    if boot_signature != 0xAA55:
        print(f"警告: 引导扇区签名不正确: 0x{boot_signature:04X} (应该是0xAA55)")
    else:
        print("✓ 引导扇区签名正确: 0xAA55")

    # 跳转指令
    jump = data[0:3]
    print(f"跳转指令: {jump.hex()} ({jump[0]:02X}")

    # OEM名称
    oem = data[3:11].decode('ascii', errors='ignore')
    print(f"OEM名称: {oem}")

    # 每扇区字节数
    bytes_per_sector = struct.unpack('<H', data[11:13])[0]
    print(f"每扇区字节数: {bytes_per_sector}")

    # 每簇扇区数
    sectors_per_cluster = data[13]
    print(f"每簇扇区数: {sectors_per_cluster}")

    # 保留扇区数
    reserved_sectors = struct.unpack('<H', data[14:16])[0]
    print(f"保留扇区数: {reserved_sectors}")

    # FAT表数量
```

```
num_fats = data[16]
print(f"FAT表数量: {num_fats}")

# 根目录条目数 (FAT32中通常为0)
root_entries = struct.unpack('<H', data[17:19])[0]
print(f"根目录条目数: {root_entries}")

# 总扇区数 (小)
total_sectors_small = struct.unpack('<H', data[19:21])[0]
print(f"总扇区数 (小): {total_sectors_small}")

# 媒体描述符
media_desc = data[21]
print(f"媒体描述符: 0x{media_desc:02X}")

# 每FAT扇区数 (FAT12/16)
sectors_per_fat_old = struct.unpack('<H', data[22:24])[0]
print(f"每FAT扇区数 (旧): {sectors_per_fat_old}")

# 每磁道扇区数
sectors_per_track = struct.unpack('<H', data[24:26])[0]
print(f"每磁道扇区数: {sectors_per_track}")

# 磁头数
num_heads = struct.unpack('<H', data[26:28])[0]
print(f"磁头数: {num_heads}")

# 隐藏扇区数
hidden_sectors = struct.unpack('<I', data[28:32])[0]
print(f"隐藏扇区数: {hidden_sectors}")

# 总扇区数 (大)
total_sectors = struct.unpack('<I', data[32:36])[0]
print(f"总扇区数 (大): {total_sectors}")

# 每FAT扇区数 (FAT32)
sectors_per_fat = struct.unpack('<I', data[36:40])[0]
print(f"每FAT扇区数 (FAT32): {sectors_per_fat}")

# 扩展标志
ext_flags = struct.unpack('<H', data[40:42])[0]
print(f"扩展标志: 0x{ext_flags:04X}")

# 文件系统版本
fs_version = struct.unpack('<H', data[42:44])[0]
print(f"文件系统版本: {fs_version}")

# 根目录簇号
root_cluster = struct.unpack('<I', data[44:48])[0]
print(f"根目录簇号: {root_cluster}")

# FSInfo扇区号
fsinfo_sector = struct.unpack('<H', data[48:50])[0]
print(f"FSInfo扇区号: {fsinfo_sector}")

# 备份引导扇区
```

```

backup_boot = struct.unpack('<H', data[50:52])[0]
print(f"备份引导扇区: {backup_boot}")

# 卷序列号
volume_serial = struct.unpack('<I', data[67:71])[0]
print(f"卷序列号: 0x{volume_serial:08X}")

# 卷标
volume_label = data[71:82].decode('ascii', errors='ignore').strip()
print(f"卷标: '{volume_label}'")

# 文件系统类型
fs_type = data[82:90].decode('ascii', errors='ignore').strip()
print(f"文件系统类型: '{fs_type}'")

print("=" * 60)

return {
    'bytes_per_sector': bytes_per_sector,
    'sectors_per_cluster': sectors_per_cluster,
    'reserved_sectors': reserved_sectors,
    'num_fats': num_fats,
    'sectors_per_fat': sectors_per_fat,
    'root_cluster': root_cluster,
    'total_sectors': total_sectors if total_sectors > 0 else
total_sectors_small,
    'fsinfo_sector': fsinfo_sector,
    'backup_boot': backup_boot,
    'volume_serial': volume_serial,
    'volume_label': volume_label
}

def analyze_fat_table(data, boot_info):
    """分析FAT表"""
    print("\n" + "=" * 60)
    print("FAT表分析")
    print("=" * 60)

    bytes_per_sector = boot_info['bytes_per_sector']
    reserved_sectors = boot_info['reserved_sectors']
    sectors_per_fat = boot_info['sectors_per_fat']
    num_fats = boot_info['num_fats']

    # FAT表起始位置
    fat_start = reserved_sectors * bytes_per_sector
    fat_size = sectors_per_fat * bytes_per_sector

    print(f"FAT表起始偏移: 0x{fat_start:x} ({fat_start} 字节)")
    print(f"每个FAT表大小: {fat_size} 字节")
    print(f"FAT表数量: {num_fats}")

    if fat_start + fat_size * num_fats > len(data):
        print(f"警告: FAT表超出数据范围!")
        return None, None

    # 读取第一个FAT表

```

```

fat1_data = data[fat_start:fat_start + fat_size]

print(f"\n第一个FAT表前64个条目:")
for i in range(0, min(64, len(fat1_data) // 4)):
    offset = i * 4
    if offset + 4 <= len(fat1_data):
        entry = struct.unpack('<I', fat1_data[offset:offset+4])[0]
        entry_val = entry & 0x0FFFFFFF # FAT32使用低28位
        status = ""
        if i == 0:
            status = " (媒体描述符)"
        elif entry_val == 0:
            status = " (空闲)"
        elif entry_val == 0x0FFFFFF7:
            status = " (坏簇)"
        elif entry_val >= 0x0FFFFFF8:
            status = " (文件结束)"
        elif entry_val >= 2:
            status = f" (指向簇 {entry_val})"

        if entry_val != 0 or i < 10:
            print(f" FAT[{i}]: 0x{entry_val:08X} ({entry_val}){status}")

# 检查FAT表是否有损坏
print(f"\n检查FAT表完整性...")
zero_entries = sum(1 for i in range(0, len(fat1_data), 4)
                   if struct.unpack('<I', fat1_data[i:i+4])[0] & 0x0FFFFFFF
== 0)
print(f" 空闲条目数: {zero_entries}")

# 如果有第二个FAT表, 比较它们
if num_fats > 1:
    fat2_start = fat_start + fat_size
    fat2_data = data[fat2_start:fat2_start + fat_size]

    if fat1_data == fat2_data:
        print(f" ✓ 两个FAT表一致")
    else:
        print(f" X 两个FAT表不一致! 可能有损坏或隐藏数据")
        # 找出差异
        diff_count = 0
        for i in range(0, min(len(fat1_data), len(fat2_data)), 4):
            if fat1_data[i:i+4] != fat2_data[i:i+4]:
                diff_count += 1
                if diff_count <= 10:
                    e1 = struct.unpack('<I', fat1_data[i:i+4])[0] &
0x0FFFFFFF
                    e2 = struct.unpack('<I', fat2_data[i:i+4])[0] &
0x0FFFFFFF
                    print(f" 条目 {i//4}: FAT1=0x{e1:08X},
FAT2=0x{e2:08X}")
                print(f" 总差异数: {diff_count}")

        return fat_start, fat_size

def analyze_root_directory(data, boot_info):

```

```

"""分析根目录"""
print("\n" + "=" * 60)
print("根目录分析")
print("=" * 60)

bytes_per_sector = boot_info['bytes_per_sector']
sectors_per_cluster = boot_info['sectors_per_cluster']
reserved_sectors = boot_info['reserved_sectors']
sectors_per_fat = boot_info['sectors_per_fat']
num_fats = boot_info['num_fats']
root_cluster = boot_info['root_cluster']

# 计算数据区起始位置
data_start = (reserved_sectors + num_fats * sectors_per_fat) *
bytes_per_sector

# 根目录起始位置（从根目录簇号计算）
cluster_size = sectors_per_cluster * bytes_per_sector
root_start = data_start + (root_cluster - 2) * cluster_size

print(f"数据区起始偏移: 0x{data_start:x} ({data_start} 字节)")
print(f"根目录起始偏移: 0x{root_start:x} ({root_start} 字节)")
print(f"簇大小: {cluster_size} 字节")
print(f"根目录簇号: {root_cluster}")

if root_start >= len(data):
    print("警告: 根目录偏移超出数据范围!")
    return

# 读取根目录条目（每个条目32字节）
root_data = data[root_start:root_start + min(cluster_size, len(data) -
root_start)]

print(f"\n根目录条目:")
entry_count = 0
for i in range(0, len(root_data) // 32):
    offset = i * 32
    entry = root_data[offset:offset+32]

    # 检查是否为空条目
    if entry[0] == 0x00:
        break
    if entry[0] == 0xE5:
        print(f"  条目 {i}: [已删除]")
        continue

    entry_count += 1

# 文件名（8字节）+ 扩展名（3字节）
filename = entry[0:8].decode('ascii', errors='ignore').strip()
ext = entry[8:11].decode('ascii', errors='ignore').strip()
fullname = f"{filename}.{ext}" if ext else filename

# 属性
attr = entry[11]
attr_str = []

```

```

if attr & 0x01: attr_str.append('只读')
if attr & 0x02: attr_str.append('隐藏')
if attr & 0x04: attr_str.append('系统')
if attr & 0x08: attr_str.append('卷标')
if attr & 0x10: attr_str.append('目录')
if attr & 0x20: attr_str.append('归档')

# 起始簇号（高16位在0x14-0x15，低16位在0x1A-0x1B）
cluster_high = struct.unpack('<H', entry[20:22])[0]
cluster_low = struct.unpack('<H', entry[26:28])[0]
cluster = (cluster_high << 16) | cluster_low

# 文件大小
file_size = struct.unpack('<I', entry[28:32])[0]

print(f"  条目 {i}: {fullname}")
print(f"  属性: 0x{attr:02x} ({', '.join(attr_str) if attr_str else '普通文件'}")
print(f"  起始簇: {cluster}")
print(f"  文件大小: {file_size} 字节")
print(f"  原始数据: {entry.hex()}")

print(f"\n总有效条目数: {entry_count}")

def search_hidden_data(data):
    """搜索隐藏的数据和异常模式"""
    print("\n" + "=" * 60)
    print("搜索隐藏信息和异常模式")
    print("=" * 60)

    # 搜索常见的CTF标志格式
    patterns = [
        (b'flag{', 'flag格式'),
        (b'FLAG{', 'FLAG格式'),
        (b'ISCTF{', 'ISCTF格式'),
        (b'isctf{', 'isctf格式'),
        (b'ctf{', 'ctf格式'),
        (b'CTF{', 'CTF格式'),
        ('静谧'.encode('utf-8'), '静谧'),
        (b'\xe9\x9d\x99', '静谧(UTF-8)'),
    ]

    found_flags = []
    for pattern, desc in patterns:
        pos = 0
        while True:
            pos = data.find(pattern, pos)
            if pos == -1:
                break
            # 尝试提取完整标志
            if pattern[0] in [b'{'[0], b'('[0]]:
                end_pos = data.find(b'}', pos) if b'}' in pattern else
data.find(b')', pos)
            else:
                end_pos = pos + len(pattern) + 50 # 尝试读取更多

```

```

        if end_pos != -1 and end_pos < len(data):
            try:
                flag = data[pos:end_pos+1].decode('ascii', errors='ignore')
                found_flags.append((pos, flag, desc))
            except:
                pass
        pos += 1

if found_flags:
    print("\n发现可能的标志:")
    for pos, flag, desc in found_flags:
        print(f" 偏移 0x{pos:X}: {flag[:100]} ({desc})")
else:
    print("未发现标准格式的标志")

# 搜索可打印ASCII字符串
print("\n搜索可打印ASCII字符串 (长度>=10, 包含关键词):")
current_str = b''
strings_found = []

for i, byte in enumerate(data):
    if 32 <= byte <= 126: # 可打印ASCII
        current_str += bytes([byte])
    else:
        if len(current_str) >= 10:
            try:
                decoded = current_str.decode('ascii')
                if any(keyword in decoded.lower() for keyword in ['flag',
'secret', 'hidden', 'key', 'password', 'isctf', 'ctf', '静谧', 'quiet']):
                    strings_found.append((i - len(current_str), decoded))
            except:
                pass
        current_str = b''

# 显示找到的字符串
for offset, s in strings_found[:20]:
    print(f" 偏移 0x{offset:X}: {s[:100]}")

def hex_dump(data, offset, length=512):
    """十六进制转储"""
    print(f"\n偏移 0x{offset:X} 的十六进制转储 ({length} 字节):")
    for i in range(0, min(length, len(data) - offset), 16):
        hex_str = ' '.join(f'{b:02X}' for b in data[offset+i:offset+i+16])
        ascii_str = ' '.join(chr(b) if 32 <= b <= 126 else '.' for b in
data[offset+i:offset+i+16])
        print(f" {offset+i:08X}: {hex_str:<48} {ascii_str}")

def main():
    filename = 'fat32_disk.img'

    if not os.path.exists(filename):
        print(f"错误: 文件 {filename} 不存在")
        print("请先运行 deep_analyze_vmdk.py 提取FAT32数据")
        return

    print(f"正在分析FAT32磁盘镜像: {filename}")

```

```

print(f"文件大小: {os.path.getsize(filename)} 字节\n")

with open(filename, 'rb') as f:
    data = f.read()

# 分析引导扇区
boot_info = read_boot_sector(data)

if boot_info is None:
    return

# 分析FAT表
fat_start, fat_size = analyze_fat_table(data, boot_info)

# 分析根目录
analyze_root_directory(data, boot_info)

# 搜索隐藏数据
search_hidden_data(data)

# 显示引导扇区的十六进制转储
hex_dump(data, 0, 512)

# 检查备份引导扇区
if boot_info['backup_boot'] > 0:
    backup_offset = boot_info['backup_boot'] * boot_info['bytes_per_sector']
    if backup_offset < len(data):
        print(f"\n检查备份引导扇区 (偏移 0x{backup_offset:x}) :")
        hex_dump(data, backup_offset, 512)

# 检查FSInfo扇区
if boot_info['fsinfo_sector'] > 0:
    fsinfo_offset = boot_info['fsinfo_sector'] *
boot_info['bytes_per_sector']
    if fsinfo_offset < len(data):
        print(f"\n检查FSInfo扇区 (偏移 0x{fsinfo_offset:x}) :")
        hex_dump(data, fsinfo_offset, 512)

if __name__ == '__main__':
    main()

```

User

静谧之眼不用管 只需要将fat区修复

Cursor

先分析 FAT 表状态, 再修复:

FAT表位置可能不正确。检查VMDK文件中FAT表的实际位置, 并修复:

```

#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
修复FAT32文件系统的FAT表
"""

import struct
import os

def read_boot_sector(data):
    """读取引导扇区信息"""
    bytes_per_sector = struct.unpack('<H', data[11:13])[0]
    sectors_per_cluster = data[13]
    reserved_sectors = struct.unpack('<H', data[14:16])[0]
    num_fats = data[16]
    sectors_per_fat = struct.unpack('<I', data[36:40])[0]
    root_cluster = struct.unpack('<I', data[44:48])[0]

    return {
        'bytes_per_sector': bytes_per_sector,
        'sectors_per_cluster': sectors_per_cluster,
        'reserved_sectors': reserved_sectors,
        'num_fats': num_fats,
        'sectors_per_fat': sectors_per_fat,
        'root_cluster': root_cluster,
    }

def find_fat_in_vmdk(vmdk_data, fat_offset_in_image):
    """在VMDK文件中查找FAT表的实际位置"""
    # FAT表在镜像中的偏移是fat_offset_in_image
    # 但在VMDK文件中，镜像从0x60000开始
    vmdk_fat_offset = 0x60000 + fat_offset_in_image

    if vmdk_fat_offset < len(vmdk_data):
        return vmdk_fat_offset
    return None

def analyze_and_fix_fat(vmdk_file, output_file):
    """分析并修复FAT表"""
    print("=" * 60)
    print("FAT32 FAT表修复工具")
    print("=" * 60)

    with open(vmdk_file, 'rb') as f:
        vmdk_data = f.read()

    # FAT32镜像从0x60000开始
    image_start = 0x60000
    if image_start >= len(vmdk_data):
        print("错误：VMDK文件太小，找不到FAT32镜像")
        return

    fat32_data = vmdk_data[image_start:]

    if len(fat32_data) < 512:

```

```

    print("错误: FAT32镜像数据太小")
    return

# 读取引导扇区
boot_info = read_boot_sector(fat32_data)

print(f"\n引导扇区信息:")
print(f"  每扇区字节数: {boot_info['bytes_per_sector']}")
print(f"  保留扇区数: {boot_info['reserved_sectors']}")
print(f"  FAT表数量: {boot_info['num_fats']}")
print(f"  每FAT扇区数: {boot_info['sectors_per_fat']}")
print(f"  根目录簇号: {boot_info['root_cluster']}")

# 计算FAT表位置
fat_start_in_image = boot_info['reserved_sectors'] *
boot_info['bytes_per_sector']
fat_size = boot_info['sectors_per_fat'] * boot_info['bytes_per_sector']

print(f"\nFAT表信息:")
print(f"  在镜像中的偏移: 0x{fat_start_in_image:X} ({fat_start_in_image} 字节)")
print(f"  每个FAT表大小: {fat_size} 字节")
print(f"  镜像实际大小: {len(fat32_data)} 字节")

# 检查FAT表是否在镜像范围内
if fat_start_in_image + fat_size * boot_info['num_fats'] > len(fat32_data):
    print(f"\n警告: FAT表超出镜像范围!")
    print(f"  需要: {fat_start_in_image + fat_size * boot_info['num_fats']} 字
节")
    print(f"  实际: {len(fat32_data)} 字节")
    print(f"  缺少: {fat_start_in_image + fat_size * boot_info['num_fats'] -
len(fat32_data)} 字节")

# 尝试在VMDK文件中查找FAT表
vmdk_fat_start = image_start + fat_start_in_image
if vmdk_fat_start < len(vmdk_data):
    print(f"\n在VMDK文件中查找FAT表 (偏移 0x{vmdk_fat_start:X})...")
    # 读取FAT表
    available_size = min(fat_size, len(vmdk_data) - vmdk_fat_start)
    fat1_data = vmdk_data[vmdk_fat_start:vmdk_fat_start + available_size]

    print(f"  读取到 {len(fat1_data)} 字节的FAT表数据")

# 检查FAT表前几个条目
print(f"\nFAT表前32个条目:")
for i in range(0, min(32, len(fat1_data) // 4)):
    offset = i * 4
    if offset + 4 <= len(fat1_data):
        entry = struct.unpack('<I', fat1_data[offset:offset+4])[0]
        entry_val = entry & 0xFFFFFFFF
        if entry_val != 0 or i < 10:
            status = ""
            if i == 0:
                status = " (媒体描述符)"
            elif entry_val == 0:
                status = " (空闲)"
            elif entry_val == 0xFFFFFFFF:

```

```

        status = " (坏簇)"
    elif entry_val >= 0x0FFFFFF8:
        status = " (文件结束)"
    print(f"    FAT[{i}]: 0x{entry_val:08X} ({entry_val})
{status}")
    else:
        print(f"错误: FAT表位置超出VMDK文件范围")
        return
    else:
        # FAT表在镜像范围内
        fat1_data = fat32_data[fat_start_in_image:fat_start_in_image + fat_size]
        print(f"\n✓ FAT表在镜像范围内")
        print(f"    读取到 {len(fat1_data)} 字节的FAT表数据")

# 检查是否有第二个FAT表
if boot_info['num_fats'] > 1:
    fat2_start_in_image = fat_start_in_image + fat_size
    vmdk_fat2_start = image_start + fat2_start_in_image

    if vmdk_fat2_start < len(vmdk_data):
        available_size = min(fat_size, len(vmdk_data) - vmdk_fat2_start)
        fat2_data = vmdk_data[vmdk_fat2_start:vmdk_fat2_start +
available_size]

        print(f"\n检查第二个FAT表 (偏移 0x{vmdk_fat2_start:X})...")
        print(f"    读取到 {len(fat2_data)} 字节")

# 比较两个FAT表
min_size = min(len(fat1_data), len(fat2_data))
if fat1_data[:min_size] == fat2_data[:min_size]:
    print(f"    ✓ 两个FAT表一致")
else:
    print(f"    ✗ 两个FAT表不一致!")
    # 找出差异
    diff_count = 0
    for i in range(0, min_size, 4):
        if fat1_data[i:i+4] != fat2_data[i:i+4]:
            diff_count += 1
            if diff_count <= 10:
                e1 = struct.unpack('<I', fat1_data[i:i+4])[0] &
0x0FFFFFFF
                e2 = struct.unpack('<I', fat2_data[i:i+4])[0] &
0x0FFFFFFF
                print(f"        条目 {i//4}: FAT1=0x{e1:08X},
FAT2=0x{e2:08X}")

    if diff_count > 0:
        print(f"    总差异数: {diff_count}")
        print(f"\n修复: 将FAT2复制为FAT1的副本...")
        # 修复FAT2
        vmdk_data_fixed = bytearray(vmdk_data)
        vmdk_data_fixed[vmdk_fat2_start:vmdk_fat2_start + min_size] =
fat1_data[:min_size]

        with open(output_file, 'wb') as f:
            f.write(vmdk_data_fixed)

```

```

        print(f" ✓ 已修复并保存到: {output_file}")
        return True

# 检查FAT表是否有明显的损坏
print(f"\n检查FAT表完整性...")

# FAT表第一个条目应该是媒体描述符 (通常是0x0FFFFFF8)
first_entry = struct.unpack('<I', fat1_data[0:4])[0] & 0x0FFFFFFF
if first_entry != 0x0FFFFFF8:
    print(f" 警告: FAT[0]应该是0x0FFFFFF8, 实际是0x{first_entry:08X}")
    # 修复第一个条目
    print(f" 修复FAT[0]...")
    vmdk_data_fixed = bytearray(vmdk_data)
    fixed_entry = struct.pack('<I', 0x0FFFFFF8)
    vmdk_fat_start = image_start + fat_start_in_image
    vmdk_data_fixed[vmdk_fat_start:vmdk_fat_start+4] = fixed_entry

# 如果有第二个FAT表, 也修复它
if boot_info['num_fats'] > 1:
    vmdk_fat2_start = image_start + fat_start_in_image + fat_size
    if vmdk_fat2_start < len(vmdk_data_fixed):
        vmdk_data_fixed[vmdk_fat2_start:vmdk_fat2_start+4] = fixed_entry

with open(output_file, 'wb') as f:
    f.write(vmdk_data_fixed)
print(f" ✓ 已修复并保存到: {output_file}")
return True

print(f"\n未发现明显的FAT表损坏")
return False

def main():
    vmdk_file = '1.vmdk'
    output_file = '1_fixed.vmdk'

    if not os.path.exists(vmdk_file):
        print(f"错误: 文件 {vmdk_file} 不存在")
        return

    print(f"正在分析并修复: {vmdk_file}")
    print(f"输出文件: {output_file}\n")

    fixed = analyze_and_fix_fat(vmdk_file, output_file)

    if fixed:
        print(f"\n{'='*60}")
        print("修复完成!")
        print(f"\n{'='*60}")
    else:
        print(f"\n可能需要进一步分析FAT表的其他问题")

if __name__ == '__main__':
    main()

```

检查第二个FAT表, 并比较两个FAT表是否一致:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
检查并修复FAT32的两个FAT表
"""

import struct
import os

def main():
    vmdk_file = '1.vmdk'
    output_file = '1_fixed.vmdk'

    with open(vmdk_file, 'rb') as f:
        vmdk_data = bytearray(f.read())

    # FAT32镜像从0x60000开始
    image_start = 0x60000
    boot_sector = vmdk_data[image_start:image_start+512]

    # 读取引导扇区信息
    bytes_per_sector = struct.unpack('<H', boot_sector[11:13])[0]
    reserved_sectors = struct.unpack('<H', boot_sector[14:16])[0]
    num_fats = boot_sector[16]
    sectors_per_fat = struct.unpack('<I', boot_sector[36:40])[0]

    fat_start_in_image = reserved_sectors * bytes_per_sector
    fat_size = sectors_per_fat * bytes_per_sector

    fat1_start = image_start + fat_start_in_image
    fat2_start = fat1_start + fat_size

    print("=" * 60)
    print("FAT表检查和修复")
    print("=" * 60)
    print(f"FAT1起始位置: 0x{fat1_start:x}")
    print(f"FAT2起始位置: 0x{fat2_start:x}")
    print(f"每个FAT表大小: {fat_size} 字节")

    # 读取两个FAT表
    available_size = min(fat_size, len(vmdk_data) - fat1_start)
    fat1_data = vmdk_data[fat1_start:fat1_start + available_size]

    available_size2 = min(fat_size, len(vmdk_data) - fat2_start)
    fat2_data = vmdk_data[fat2_start:fat2_start + available_size2]

    print(f"\nFAT1实际大小: {len(fat1_data)} 字节")
    print(f"FAT2实际大小: {len(fat2_data)} 字节")

    # 比较两个FAT表
    min_size = min(len(fat1_data), len(fat2_data))
    if fat1_data[:min_size] == fat2_data[:min_size]:
        print(f"\n√ 两个FAT表一致")
```



```

        current_str = b''

# 检查FAT条目中的异常值
print(f"\n检查FAT条目中的异常值...")
for fat_num, fat_data in enumerate([fat1_data, fat2_data], 1):
    suspicious_entries = []
    for i in range(0, min(1000, len(fat_data) // 4), 4):
        entry = struct.unpack('<I', fat_data[i:i+4])[0] & 0xFFFFFFFF
        # 检查是否有异常大的值（可能是隐藏数据）
        if entry > 0xFFFFFFFF or (entry > 1000 and entry < 0xFFFFFFFF0):
            # 检查这个值是否可能是ASCII字符的组合
            entry_bytes = struct.pack('<I', entry)
            if all(32 <= b <= 126 for b in entry_bytes[:3]):
                try:
                    ascii_str = entry_bytes[:3].decode('ascii')
                    suspicious_entries.append((i // 4, entry, ascii_str))
                except:
                    pass

    if suspicious_entries:
        print(f"\nFAT{fat_num}中的可疑条目:")
        for entry_num, entry_val, ascii_str in suspicious_entries[:20]:
            print(f"  条目 {entry_num}: 0x{entry_val:08X} (可能包含:
{ascii_str})")

if __name__ == '__main__':
    main()

```

FAT2 缺失（大小为 0），这可能是问题所在。修复：将 FAT1 复制到 FAT2 位置。

```

#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
修复缺失的FAT2表
"""

import struct
import os

def main():
    vmdk_file = '1.vmdk'
    output_file = '1_fixed.vmdk'

    print("=" * 60)
    print("修复FAT2表")
    print("=" * 60)

    with open(vmdk_file, 'rb') as f:
        vmdk_data = bytearray(f.read())

    # FAT32镜像从0x60000开始
    image_start = 0x60000
    boot_sector = vmdk_data[image_start:image_start+512]

```

```

# 读取引导扇区信息
bytes_per_sector = struct.unpack('<H', boot_sector[11:13])[0]
reserved_sectors = struct.unpack('<H', boot_sector[14:16])[0]
num_fats = boot_sector[16]
sectors_per_fat = struct.unpack('<I', boot_sector[36:40])[0]

fat_start_in_image = reserved_sectors * bytes_per_sector
fat_size = sectors_per_fat * bytes_per_sector

fat1_start = image_start + fat_start_in_image
fat2_start = fat1_start + fat_size

print(f"FAT1起始位置: 0x{fat1_start:X}")
print(f"FAT2起始位置: 0x{fat2_start:X}")
print(f"每个FAT表大小: {fat_size} 字节")

# 读取FAT1
available_size = min(fat_size, len(vmdk_data) - fat1_start)
fat1_data = vmdk_data[fat1_start:fat1_start + available_size]

print(f"\nFAT1实际大小: {len(fat1_data)} 字节")

# 检查FAT2位置
if fat2_start >= len(vmdk_data):
    print(f"\n警告: FAT2位置超出文件范围!")
    print(f" 需要扩展文件到: 0x{fat2_start + fat_size:X}")
    # 扩展文件
    vmdk_data.extend(b'\x00' * (fat2_start + fat_size - len(vmdk_data)))
    print(f" 已扩展文件到: {len(vmdk_data)} 字节")
else:
    fat2_available = len(vmdk_data) - fat2_start
    print(f"FAT2可用空间: {fat2_available} 字节")

# 将FAT1复制到FAT2
print(f"\n将FAT1复制到FAT2位置...")
copy_size = min(len(fat1_data), len(vmdk_data) - fat2_start)
vmdk_data[fat2_start:fat2_start + copy_size] = fat1_data[:copy_size]

print(f" 已复制 {copy_size} 字节")

# 保存修复后的文件
with open(output_file, 'wb') as f:
    f.write(vmdk_data)

print(f"\n✓ 已修复并保存到: {output_file}")

# 验证
print(f"\n验证修复结果...")
with open(output_file, 'rb') as f:
    fixed_data = f.read()

fat1_check = fixed_data[fat1_start:fat1_start + copy_size]
fat2_check = fixed_data[fat2_start:fat2_start + copy_size]

if fat1_check == fat2_check:
    print(f"✓ FAT2已成功修复为FAT1的副本")

```

```

else:
    print(f"X 验证失败")

# 检查修复后的文件是否可以找到隐藏信息
print(f"\n检查修复后的文件中是否有隐藏信息...")

# 搜索标志格式
patterns = [
    (b'ISCTF{', 'ISCTF格式'),
    (b'isctf{', 'isctf格式'),
    (b'FLAG{', 'FLAG格式'),
    (b'flag{', 'flag格式'),
]

for pattern, desc in patterns:
    pos = fixed_data.find(pattern)
    if pos != -1:
        end_pos = fixed_data.find(b'}', pos)
        if end_pos != -1:
            flag = fixed_data[pos:end_pos+1].decode('ascii', errors='ignore')
            print(f" 发现{desc}在偏移 0x{pos:x}: {flag}")

if __name__ == '__main__':
    main()

```

FAT2 已修复。分析修复后的文件系统，查找隐藏信息：

```

#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
从修复后的FAT32文件系统中提取文件
"""

import struct
import os

def read_boot_sector(data):
    """读取引导扇区信息"""
    bytes_per_sector = struct.unpack('<H', data[11:13])[0]
    sectors_per_cluster = data[13]
    reserved_sectors = struct.unpack('<H', data[14:16])[0]
    num_fats = data[16]
    sectors_per_fat = struct.unpack('<I', data[36:40])[0]
    root_cluster = struct.unpack('<I', data[44:48])[0]

    return {
        'bytes_per_sector': bytes_per_sector,
        'sectors_per_cluster': sectors_per_cluster,
        'reserved_sectors': reserved_sectors,
        'num_fats': num_fats,
        'sectors_per_fat': sectors_per_fat,
        'root_cluster': root_cluster,
    }

```

```

def read_fat_entry(fat_data, cluster):
    """读取FAT表中的条目"""
    if cluster * 4 + 4 > len(fat_data):
        return 0xFFFFFFFF
    entry = struct.unpack('<I', fat_data[cluster * 4:cluster * 4 + 4])[0]
    return entry & 0xFFFFFFFF

def read_cluster_chain(fat_data, start_cluster, cluster_size):
    """读取簇链"""
    clusters = []
    current = start_cluster

    while current < 0xFFFFFFFF:
        if current == 0 or current == 0xFFFFFFFF:
            break
        clusters.append(current)
        current = read_fat_entry(fat_data, current)
        if len(clusters) > 10000: # 防止无限循环
            break

    return clusters

def read_directory_entry(entry_data):
    """读取目录条目"""
    if entry_data[0] == 0x00 or entry_data[0] == 0xE5:
        return None

    # 短文件名
    filename = entry_data[0:8].decode('ascii', errors='ignore').strip()
    ext = entry_data[8:11].decode('ascii', errors='ignore').strip()
    fullname = f"{filename}.{ext}" if ext else filename

    attr = entry_data[11]
    cluster_high = struct.unpack('<H', entry_data[20:22])[0]
    cluster_low = struct.unpack('<H', entry_data[26:28])[0]
    cluster = (cluster_high << 16) | cluster_low
    file_size = struct.unpack('<I', entry_data[28:32])[0]

    return {
        'name': fullname,
        'attr': attr,
        'cluster': cluster,
        'size': file_size,
        'is_dir': bool(attr & 0x10),
        'is_hidden': bool(attr & 0x02),
    }

def extract_file(vmdk_data, boot_info, fat_data, entry, output_dir):
    """提取文件"""
    if entry['cluster'] < 2:
        return

    cluster_size = boot_info['sectors_per_cluster'] *
boot_info['bytes_per_sector']
    data_start = (boot_info['reserved_sectors'] + boot_info['num_fats'] *
boot_info['sectors_per_fat']) * boot_info['bytes_per_sector']

```

```

clusters = read_cluster_chain(fat_data, entry['cluster'], cluster_size)

file_data = b''
for cluster in clusters:
    cluster_offset = data_start + (cluster - 2) * cluster_size
    if cluster_offset + cluster_size <= len(vmdk_data):
        file_data += vmdk_data[cluster_offset:cluster_offset + cluster_size]
    else:
        break

# 截断到实际文件大小
file_data = file_data[:entry['size']]

# 保存文件
safe_name = entry['name'].replace(' ', '_').replace('.', '_')
output_path = os.path.join(output_dir, safe_name)

with open(output_path, 'wb') as f:
    f.write(file_data)

print(f" 提取: {entry['name']} -> {output_path} ({len(file_data)} 字节)")

def main():
    vmdk_file = '1_fixed.vmdk'
    output_dir = 'extracted_files'

    print("=" * 60)
    print("从修复后的FAT32文件系统提取文件")
    print("=" * 60)

    if not os.path.exists(vmdk_file):
        print(f"错误: 文件 {vmdk_file} 不存在")
        print("请先运行 repair_fat2.py 修复FAT表")
        return

    os.makedirs(output_dir, exist_ok=True)

    with open(vmdk_file, 'rb') as f:
        vmdk_data = f.read()

    # FAT32镜像从0x60000开始
    image_start = 0x60000
    fat32_data = vmdk_data[image_start:]

    if len(fat32_data) < 512:
        print("错误: FAT32数据太小")
        return

    boot_info = read_boot_sector(fat32_data)

    # 读取FAT表
    fat_start = boot_info['reserved_sectors'] * boot_info['bytes_per_sector']
    fat_size = boot_info['sectors_per_fat'] * boot_info['bytes_per_sector']
    fat_data = fat32_data[fat_start:fat_start + fat_size]

```

```

print(f"\nFAT表大小: {len(fat_data)} 字节")

# 读取根目录
data_start = (boot_info['reserved_sectors'] + boot_info['num_fats'] *
boot_info['sectors_per_fat']) * boot_info['bytes_per_sector']
cluster_size = boot_info['sectors_per_cluster'] *
boot_info['bytes_per_sector']
root_start = data_start + (boot_info['root_cluster'] - 2) * cluster_size

print(f"根目录起始: 0x{root_start:x}")

if root_start >= len(fat32_data):
    print("警告: 根目录超出数据范围")
    # 尝试在VMDK文件中查找
    root_start_vmdk = image_start + root_start
    if root_start_vmdk < len(vmdk_data):
        root_data = vmdk_data[root_start_vmdk:root_start_vmdk + cluster_size]
    else:
        print("错误: 无法找到根目录")
        return
else:
    root_data = fat32_data[root_start:root_start + cluster_size]

# 读取根目录条目
print(f"\n读取根目录条目...")
entries = []
for i in range(0, len(root_data), 32):
    entry_data = root_data[i:i+32]
    entry = read_directory_entry(entry_data)
    if entry:
        entries.append(entry)
        print(f" 发现: {entry['name']} (簇: {entry['cluster']}, 大小:
{entry['size']} 字节, {'目录' if entry['is_dir'] else '文件'}, {'隐藏' if
entry['is_hidden'] else '可见'})")

# 提取文件
print(f"\n提取文件...")
for entry in entries:
    if not entry['is_dir'] and entry['size'] > 0:
        extract_file(vmdk_data, boot_info, fat_data, entry, output_dir)

# 搜索隐藏信息
print(f"\n搜索隐藏信息...")
patterns = [
    (b'ISCTF{', 'ISCTF格式'),
    (b'isctf{', 'isctf格式'),
    (b'FLAG{', 'FLAG格式'),
    (b'flag{', 'flag格式'),
]

for pattern, desc in patterns:
    pos = vmdk_data.find(pattern)
    if pos != -1:
        end_pos = vmdk_data.find(b'}', pos)
        if end_pos != -1:
            flag = vmdk_data[pos:end_pos+1].decode('ascii', errors='ignore')

```

```
print(f" 发现{desc}在偏移 0x{pos:x}: {flag}")
```

```
if __name__ == '__main__':  
    main()
```