How OpenBSD’s malloc helps the developer

Otto Moerbeek

EuroBSDCon 2023, Coimbra Portugal

Me?

- Otto Moerbeek, OpenBSD developer since 2003, `otto@`
- DAYJOB="PowerDNS Senior Developer"
- Worked on many things, mainly user land
- Reimplemented malloc, part of OpenBSD since

Revision 1.92 / (**download**) - annotate - [select for diffs], *Mon Jul 28 04:56:38 2008 UTC* (15 years, 1 month ago) by *otto*
Branch: **MAIN**
CVS Tags: **OPENBSD_4_4_BASE**, **OPENBSD_4_4**
Changes since **1.91: +840 -1448 lines**
Diff to previous 1.91 (colored)

```
Almost complete rewrite of malloc, to have a more efficient data
structure of tracking pages returned by mmap(). Lots of testing by
lots of people, thanks to you all.
ok djm@ (for a slighly earlier version) deraadt@
```

`malloc(3)` API

- `void* malloc(size_t size);`
- `void free(void *ptr);`
- All other functions (`realloc()`, `calloc()`, etc) can be expressed in terms of the two above
- A few extra rules, e.g. about alignment
- Simple API leaves *many* opportunities to implement it in different ways
- `malloc()` ***has*** to store meta-data, at least for the size of an allocation

Some implementation choices

- How do we get memory from kernel (`sbrk(2)`, `mmap(2)`, mixed)?
- Where do we store meta-data (as part of allocated data, or separately?)
- Do we return free memory to kernel using `munmap(2)` (why would you do/not do that?)

Size matters… on OpenBSD

- `malloc()` always gets memory from kernel using `mmap(2)`.
- Minimum size of that is 1 page (typically 4k)
- `mmap(2)` in OpenBSD is *randomised.* ASLR is extended to application heap.
- For smaller allocations, `malloc()` allocates a page and divides it into *chunks*
- Per chunk page a bitmap is maintained to store which chunks are free. This is another piece of meta-data.

Design goals of OpenBSD’s malloc

- Do strict internal consistency checking
- Implement security relevant features: e.g. randomisation in many places.
- Always store meta-data out-of-band
- Try to detect API-misuse (e.g. double free)
- Help the developer to find bugs like out-of-bound-write or use-after free

Features and design choices

Feature	Typical Other	OpenBSD
Memory layout	Compact	Scattered
Return memory to kernel after `free()`	Rare	Often
Store meta-data near allocations	Yes	Never
Internal consistency checks	Few	Many
Randomisation of cache	Some	Always, for many cases
Additional optional checks	Maybe	Quite a few
Continue on error	Often	Never
Details of errors	Sometimes	Often
Speed	Fast/Ultra Fast	Varies

A program with at least one bug

```
otto@h2:~$ cat m.c
#include <stdlib.h>
#include <stdio.h>

int
main(int argc, char *argv[])
{
        size_t sz = atoi(argv[1]);
        char *p = malloc(sz);
        printf("%p\n", p);
        p[sz] = 0;
        free(p);
        return 0;
}
otto@h2:~$
```

No crash????

Lets try another system on next slide.....

```
h2$ cat m.c
#include <stdlib.h>
#include <stdio.h>

int
main(int argc, char *argv[])
{
        size_t sz = atoi(argv[1]);
        char *p = malloc(sz);
        printf("%p\n", p);
        p[sz] = 0;
        free(p);
        return 0;
}
h2$ ./m 40960
0x824533200
h2$ 
```

That’s better!

```
h1$ cat m.c
#include <stdlib.h>
#include <stdio.h>

int
main(int argc, char *argv[])
{
        size_t sz = atoi(argv[1]);
        unsigned char *p = malloc(sz);
        printf("%p\n", p);
        p[sz] = 0;
        free(p);
        return 0;
}
h1$ ./m 40960
0x14d688e000
Segmentation fault (core dumped)
h1$ 
```

Explanation

- First system is an amd64 FreeBSD system (Debian using glibc acts the same)
- Second system aarch64 OpenBSD
- In the FreeBSD case, the allocation is surrounded by *mapped* memory
- On OpenBSD, the allocation is surrounded by *unmapped* memory

What happens for smaller allocations

- On OpenBSD, a small allocation is expected to be surrounded by other chunks, as they share a page
- So we expect no immediate crash on a typical out-of-bound write for a small allocation, as it will end up in the next one
- Only segmentation violation if it was the last chunk on a page *and* the out-of-bounds write extends beyond the page

On the FreeBSD system, we see no issue with a 1000 bytes allocation

```
h2$ cat m.c
#include <stdlib.h>
#include <stdio.h>

int
main(int argc, char *argv[])
{
        size_t sz = atoi(argv[1]);
        char *p = malloc(sz);
        printf("%p\n", p);
        p[sz] = 0;
        free(p);
        return 0;
}
h2$ ./m 40960
0x824533200
h2$ ./m 1000
0x824adb000
h2$
```

- On the OpenBSD system, also no problem.
- In both cases an out-of-bound write happens.
- The memory is *mapped*, it is *malloc-owned*, not *application owned*

```
h1$ cat m.c
#include <stdlib.h>
#include <stdio.h>

int
main(int argc, char *argv[])
{
        size_t sz = atoi(argv[1]);
        unsigned char *p = malloc(sz);
        printf("%p\n", p);
        p[sz] = 0;
        free(p);
        return 0;
}
h1$ ./m 40960
0x14d688e000
Segmentation fault (core dumped)
h1$ ./m 1000
0x163d04f800
h1$
```

Adding a malloc flag on OpenBSD detects the bug

```
h1$ cat m.c
#include <stdlib.h>
#include <stdio.h>

int
main(int argc, char *argv[])
{
        size_t sz = atoi(argv[1]);
        unsigned char *p = malloc(sz);
        printf("%p\n", p);
        p[sz] = 0;
        free(p);
        return 0;
}
h1$ ./m 40960
0x14d688e000
Segmentation fault (core dumped)
h1$ ./m 1000
0x163d04f800
h1$ MALLOC_OPTIONS=C ./m 1000
0x219a4bb000
m(88952) in free(): canary corrupted 0x219a4bb000 0x3e8@0x3e8
Abort trap (core dumped)
h1$
```

glibc (Debian) has flags too, but `MALLOC_CHECK_` does not detect more issues, they only print different info on error

```
otto@h2:~$ cat m.c
#include <stdlib.h>
#include <stdio.h>

int
main(int argc, char *argv[])
{
	size_t sz = atoi(argv[1]);
	char *p = malloc(sz);
	printf("%p\n", p);
	p[sz] = 0;
	free(p);
	return 0;
}
otto@h2:~$ ./m 40960
0xaaaad0a482a0
otto@h2:~$ ./m 1000
0xaaaafc4dc2a0
otto@h2:~$ MALLOC_CHECK_=3 ./m 1000
0xaaaad7f3b2a0
otto@h2:~$
```

Canary check

- Write byte pattern after the application owned allocation if the malloc owned allocation is larger that the application owned

- On `free()`, check if the canary was overwritten

- Enabled with malloc option `C` (included in `S`)

A double-free case

On Debian (and FreeBSD), the chunk is re-used.

Even withchecking by malloc, this will *not* get caught, the second call to `free` actually is “fine”.

```
otto@h2:~$ cat m2.c
#include <stdlib.h>
#include <stdio.h>

int
main(int argc, char *argv[])
{
        size_t sz = atoi(argv[1]);
        unsigned char *p = malloc(sz), *q;
        printf("%p\n", p);
        free(p);
        q = malloc(sz);
        printf("%p\n", q);
        free(p);
        return 0;
}
otto@h2:~$ MALLOC_CHECK_=3 ./m2 2000
0xaaaad2bb82a0
0xaaaad2bb82a0
otto@h2:~$
```

On OpenBSD, some runs catch the error.

This is randomisation in action, plus *delayed free list*.

With malloc option F it is always caught

```
h1$ ./m2 2000
0x961352800
0x961352000
h1$ ./m2 2000
0xb7cefd800
0xb7ced7800
h1$ ./m2 2000
0xcefef0800
0xcefeb4800
h1$ ./m2 2000
0x1581e51000
0x1581e45800
m2(54772) in free(): double free 0x1581e51000
Abort trap (core dumped)
h1$ MALLOC_OPTIONS=F ./m2 2000
0x1e353d2000
0x1e353bd800
m2(61396) in free(): double free 0x1e353d2000
Abort trap (core dumped)
h1$
```

Re-using allocations

- Not doing it has big performance impact
- Immediately doing it has big potential impact: heap usage errors can turn into security bugs. OpenBSD uses delayed free list to limit impact: chunks are never *immediately* re-used.
- For page-sized allocations we have a cache for performance reasons, re-use is randomised and it is *completely* disabled with malloc option S
- Double-free checks are done, but due to randomisation not triggered always
- Sometimes confusing: errors may be detected for allocation X while freeing allocation Y.
- More extensive double-free checks are done wit malloc option F (included in S)

Leak detection

- Leaks are bad, but not an API usage error
- As OpenBSD's malloc stored all meta-data out-of-band, it can use meta-data to list leaks
- Function has been available for a long time
- Actually using the feature was cumbersome

Original solution

- Not compiled in by default
- Used file write to dump information if malloc option `D` was active and a file `malloc.out` existed in the current working dir
- It was a nuisance having to recompile libc to use it
- Does not work with pledged programs: often not able to write files

New solution

- Always compiled in
- Export data using `utrace(2)`
- Use `ktrace(8)` to collect and `kdump(2)` to display information
- Some flexibility to record callers

- Run with `malloc` option D
- Use `ktrace` to collect `utrace` records
- Display with `kdump`

```
int
main(int argc, char *argv[])
{
        size_t i, sz = atoi(argv[1]);

        void **p = malloc(sz * sizeof(void *));
        for (i = 0; i < sz; i++)
                p[i] = malloc(sz);
        for (i = 1; i < sz; i++)
                free(p[i]);
        return 0;
}
h1$ MALLOC_OPTIONS=D ktrace -tu ./m3 10000
h1$ kdump -u malloc
******** Start dump m3 *******
M=8 I=1 F=0 U=0 J=1 R=0 X=0 C=0 cache=64 G=0
Leak report:
                 f     sum      #    avg
      0x189b7a0a98   80000      1  80000 addr2line -e ./m3 0x10a98
      0x189b7a0abc   10000      1  10000 addr2line -e ./m3 0x10abc

******** End dump m3 *******
h1$
```

Use `addr2line` to display not-freed allocations

```
        size_t i, sz = atoi(argv[1]);

        void **p = malloc(sz * sizeof(void *));
        for (i = 0; i < sz; i++)
                p[i] = malloc(sz);
        for (i = 1; i < sz; i++)
                free(p[i]);
        return 0;
}
h1$ MALLOC_OPTIONS=D ktrace -tu ./m3 10000
h1$ kdump -u malloc
******** Start dump m3 *******
M=8 I=1 F=0 U=0 J=1 R=0 X=0 C=0 cache=64 G=0
Leak report:
                 f     sum      #    avg
      0x189b7a0a98   80000      1  80000 addr2line -e ./m3 0x10a98
      0x189b7a0abc   10000      1  10000 addr2line -e ./m3 0x10abc

******** End dump m3 *******
h1$ addr2line -e ./m3 0x10a98
/home/otto/m3.c:9
h1$ addr2line -e ./m3 0x10abc
/home/otto/m3.c:11
h1$
```

How does it work?

- On call to malloc the caller is saved using `__builtin_return_address(depth)` and `__builtin_extract_return_addr(p);`

- Sadly the docs say:
  "On some machines it may be impossible to determine the return address of any function other than the current one; in such cases, or when the top of the stack has been reached, this function returns an unspecified value."

- Runtime cost is very low: just an extra pointer stored per large allocation, or one pointer per page used for chunks

Continued…

- After the program finishes, an `atexit()` handler walks the meta data
- It will aggregate all non-freed allocations having the same caller.
- It dumps the information, including an `addr2line` line with `f` compensated for library/executable offset.

Chunks

- To save memory used for meta-data, not all allocations are recorded
- Only the ones that end up in slot 0 of a chunk page
- Run several times to get non-zero `f` values.

Why not more features?

- Run time overhead even if not actively used
- Avoid too complex code
- A middle ground solution: always available, but not *very* fancy functionality. For more thorough heap debugging other tools can be used
- Example of tool: `valgrind`, though it does not work very well on OpenBSD (yet) (sad trombone…)

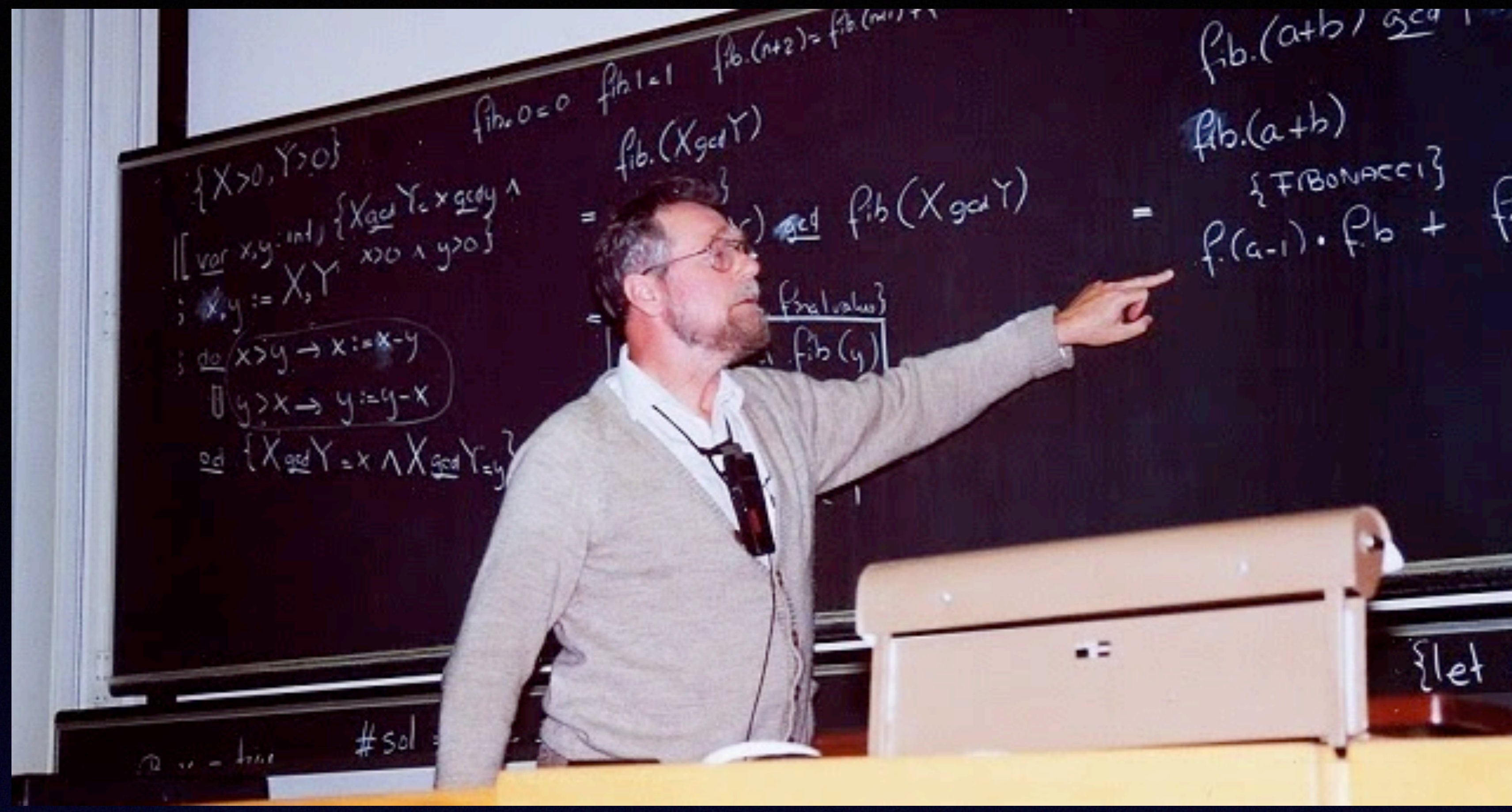
- Debian run
- Notice it shows only one leak, only hints at the other
- Full stack trace instead of only caller
- Full history of allocation is captured: allocation point, point of free and out-of-bound accesses

```
==5981== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==5981== Command: ./m3 10000
==5981==
==5981==
==5981== HEAP SUMMARY:
==5981==     in use at exit: 90,000 bytes in 2 blocks
==5981==   total heap usage: 10,001 allocs, 9,999 frees, 100,080,000 bytes alloc
ated
==5981==
==5981== 90,000 (80,000 direct, 10,000 indirect) bytes in 1 blocks are definitel
y lost in loss record 2 of 2
==5981==    at 0x48850C8: malloc (vg_replace_malloc.c:381)
==5981==    by 0x10884B: main (m3.c:9)
==5981==
==5981== LEAK SUMMARY:
==5981==    definitely lost: 80,000 bytes in 1 blocks
==5981==    indirectly lost: 10,000 bytes in 1 blocks
==5981==      possibly lost: 0 bytes in 0 blocks
==5981==    still reachable: 0 bytes in 0 blocks
==5981==         suppressed: 0 bytes in 0 blocks
==5981==
==5981== For lists of detected and suppressed errors, rerun with: -s
==5981== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
otto@h2:~$
```

Back to OpenBSD: malloc options

- `F` Freecheck
- `J` More junking
- `C` Canary checks
- `U` Free unmap
- Most important: `S`

malloc helps



- Strictness not only useful to avoid security issues
- Randomisation: each run is different, catching bugs that depend on specific memory layout
- Add to that other checks and malloc flags, OpenBSD's malloc helps as a strict (but fair!) teacher to get your heap usage in order.
- During development and bug hunting, use malloc option `S`!
- Check your program with malloc option `D` for leaks