Giải thích sự tồn tại của random brk offset giữa Heap và BSS segment, vì sao có khoảng gap này?

1. Giới thiệu

Như ta đã biết, memory segment được chia thành các phần textsegment, data segment, bss segment, heap, stack. Nếu việc tính toán xảy ra một cách bình thường và trơn tru nhất có thể, địa chỉ ảo của các thành phần trên đều giống nhau đối với mọi process thực thi trên 1 máy tính. Các kẻ tấn công từ xa có thể đi vào các thành phần địa chỉ của thư viện hoặc binary một cách đơn giản và phá hoại bằng các lỗi hay gây overload trong hệ thống máy, bởi vì các địa chỉ đều giống nhau. Do đó, việc thêm các không gian ngẫu nhiên giữa các đoạn của bộ nhớ đang dần trở nên phổ biến và cơ chế này được gọi là Address Space Layout Randomization (ASLR).

Trên Linux, điều này được thực hiện bằng cách thêm các offset ngẫu nhiên vào địa chỉ bắt đầu của stack, memory mapping segment, heap. Điều đó tạo ra khoảng random brk offset giữa heap và BSS segment.

Linux còn. cho phép 3 chế độ ASLR khác nhau được lựa chọn:

1. Không ngẫu nhiện, mọi địa chỉ trong các process đều bắt đầu như nhau
2. Ngẫu nhiên một phần, bao gồm các thư viện chung, stack, mmap(), heap, virtual dynamic shared object.
3. Tất cả đều ngẫu nhiên.

Bên cạnh đóm Linux còn cho phép thêm một cơ chế gọi là position independent executable (PIE) binary. PIE là một các ngẫu nhiên không gian địa chỉ ngẫu nhiên được thêm vào để compile và liên kết các chương trình thực thi để khiến chúng hoàn toàn độc lập. Điều này khiến cho các binary được compile có code segment, global offset table và procedure linkage table được đặt vào các vị trí hoàn toàn ngẫu nhiên mỗi lần trong bộ nhớ ảo mỗi lần ứng dụng được thực thi.

2. Hiện thực ASLR trên nhân Linux

Hiện thực của hàm radomize_page() để tạo ra một địa chỉ ngẫu nhiên có độ dài len nằm giữa [start; end]. Hiện thực của hàm này có thể được tìm thấy trên /drivers/char/random.c

```
unsigned long
randomize_range(unsigned long start, unsigned long end, unsigned long len)
{
        unsigned long range = end - len - start;

        if (end <= start + len)
                return 0;
        return PAGE_ALIGN(get_random_int() % range + start);
}
```

Từ đó, Linux hiện thực thêm hàm arch_randomize_brk() để thêm vào một offset ngẫu nhiên vào đầu của heap. Điều đó khiến cho có khoảng trống giữa BSS và heap trong code segment.

```
unsigned long arch_randomize_brk(struct mm_struct *mm)
{
         unsigned long range_end = mm->brk + 0x02000000;
         return randomize_range(mm->brk, range_end, 0) ? : mm->brk;
}
```

Ngoài ra, cũng còn rất nhiều cơ chế thêm ngẫu nhiên khác có thể được tìm thấy trong /arch/x86/kernel/process.c

3. Hạn chế

Tuy việc thêm các offset ngẫu nhiên có thể bảo đảm an toàn cho hệ thống máy tính của bạn, việc chiếm một số lượng các bit để thực hiện các công việc ngẫu nhiên rõ ràng chiếm tương

đối nhiều không gian, không gian còn lại để thực hiện việc định bộ nhớ là không nhiều, đặt biệt trong các hệ thống 32-bit.
Theo thông tin, trên các hệ thống 32-bit, người ra thường dùng 16 đến để thực hiện tạo offset ngẫu nhiên, trên 64-bit thường không dưới 40-bit.
Vấn đề phân mảnh hiện diện rất rõ đối với 32-bit. Việc random vào bộ nhớ và cách đoạn làm cho bộ nhớ có những khoảng trắng không được sử dụng và không được truy cập. Đến một lúc nào đó, sẽ không còn một khoảng trống khả dụng để thêm vào 1 process nữa. Với bộ nhớ 64-bit lớn hơn, điều này khó để ý hơn.
Ngoài ra, với việc sử dụng chỉ 16-bit bộ nhớ, điều đó có nghĩa là có 16_{16} vị trí có thể ngẫu nhiên, điều này khiến cho việc tấn công vào bằng brute force, vốn có thể thực hiện bằng nhiều hệ thống máy tính ngày nay. Với 40-bit ngẫu nhiên, hệ thống 64-bit sẽ khó bị tấn công hơn, trừ khi bị leak dữ liệu.
Một điều có thể để ý, hàm get_random_int() hay nhiều hàn random khác trong hiện tại vẫn chưa hoàn toàn tạo ra một số random mà chỉ là theo một công thức nào đó. Nếu bị đoán ra quy luật, ta hoàn toàn có thể vượt qua rào cản này, truy cập trực tiếp vào bộ nhớ và thực hiện các cuộc tấn công.
Ngoài ra, do là sản phẩm của con người nên luôn tồn tại các lỗi trong nhiều trường hợp, qua đó tin tặc có thể qua mặt các lớp bảo mật này.

Tham khảo

[1] https://www.sciencedirect.com/topics/computer-science/address-space-layout-randomization
[2] https://web.stanford.edu/~blp/papers/asrandom.pdf
[3] https://github.com/0xricksanchez/articles/blob/master/ASLR/README.md