BleedingTooth

블루투스, 어디까지 해킹할 수 있을까?

31기 육은서, 함은지, 황선영

<목차>

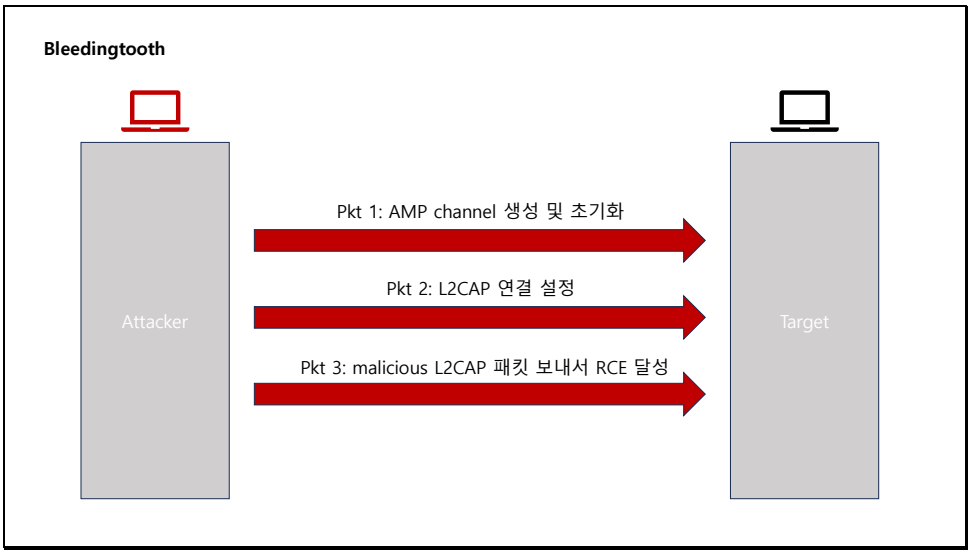
I. 블루투스 취약점의 한계와 블리딩투스의 의의

해당 칼럼에서는 블루투스 호스트 공격인 Bleedingtooth에 대해 설명할 예정이다. Bleedingtooth는 제로클릭 원격 코드 실행 공격이다.

보통 블루투스 공격은 펌웨어 타겟이거나 정보를 도청하고 조작하는 수준으로 이루어진다. 하지만, Bleedingtooth는 타겟 디바이스를 완전히 제어할 수 있다는 점에서 의의를 갖는다.

II. 블루투스 동작 방식

Bleedingtooth의 자세한 attack surface는 BlueZ이다. BlueZ란 리눅스 상에서 블루투스 기능을 구현하기 위한 라이브러리이다. 공격자가 타겟의 BD 주소[1]를 알고 있을 때, 조작된 l2cap 패킷을 보내서 RCE를 이뤄낼 수 있다. 이후에 나올 취약점과 익스플로잇 설명을 위해 해당 단락에서 l2cap 프로토콜에 대해 짚고 넘어갈 것이다.



프로토콜을 이해할 뿐 아니라 아래서부터는 익스플로잇에 대해 자세히 살펴볼 예정이므로 간략한 공격방법을 여기서 설명해야 한다. 위처럼 malicious 패킷을 보내면 받은 패킷을 처리하는 과정에서 취약점이 발생하게 되고, RCE가 달성된다. 이때 L2CAP는 블루투스 프로토콜 스택 중에 한 계층이다. 블루투스는 host(PC)와 controller(bluetooth 칩, 드라이버 등) 사이에 HCI(host controller interface)를 두어 데이터를 전송한다. 이때 HCI가 각 계층의 데이터를 원활하게 전송하기 위해 L2CAP라는 프로토콜을 이용한다. L2CAP는 채널 기반으로 작동하는데 bleedingtooth에서 사용하는 채널은 AMP이다(아래서는 a2mp로 언급된다)

[1] Bluetooth Device address, 블루투스 장치마다 갖고 있는 고유 장치 주소

III. BadChoice

3번, 4번은 Bleedingtooth에서 사용하는 취약점을 설명한다. 먼저 BadChoice는 CVE-2020-12352로, memory leak이 가능한 취약점이다. 먼저 A2MP_GETINFO_REQ의 함수를 살펴보자.

```
1. static int a2mp_getinfo_req(struct amp_mgr *mgr, struct sk_buff *skb,
2.                              struct a2mp_cmd *hdr)
3. {
4.       struct a2mp_info_req *req  = (void *) skb->data;
5.       ...
6.       hdev = hci_dev_get(req->id);
7.       if (!hdev || hdev->dev_type != HCI_AMP) {
8.                struct a2mp_info_rsp rsp;
9.
10.               rsp.id = req->id;
11.               rsp.status = A2MP_STATUS_INVALID_CTRL_ID;
12.
13.               a2mp_send(mgr, A2MP_GETINFO_RSP, hdr->ident, sizeof(rsp), &rsp);
14.
15.               goto done;
16.      }
17.      ...
18. }
```

If문 조건일 시, a2mp_info_rsp 타입 변수를 선언하고 초기화 후 send하는 함수이다. 다만, a2mp_info_rsp 구조체는 id, status뿐 아니라 더 많은 멤버들이 존재한다. 완전히 초기화되지 않기 때문에 leak되는 메모리가 존재한다.

IV. BadKarma

BadKarma로 불리는 CVE-2020-12351은 Type confusion 취약점이다. 이 취약점도 간단하게만 설명하자면 sock 타입의 인수를 받는 함수에 amp_mgr이 들어감으로써 생기는 confusion이다.

```
1. static int l2cap_data_rcv(struct l2cap_chan *chan, struct sk_buff *skb) {
2.     ...
3.     if ((chan->mode == L2CAP_MODE_ERTM || chan->mode == L2CAP_MODE_STREAMING)
           && sk_filter(chan->data, skb))
4.         goto drop;
5.     ...
6. }
```

L2CAP에서 streaming 혹은 ERTM 모드가 사용될 때 sk_filter가 호출된다. 이때 호출하는 인자가 chan-> data이다. chan은 l2cap_chan으로 선언되어 있다.

```
static inline int sk_filter(struct sock *sk, struct sk_buff *skb) {
        return sk_filter_trim_cap(sk, skb, 1);
}
```

sk_filter 선언부분을 보면 sock 구조체를 인수로 받고 있다. 이부분에서 type confusion이 일어난다.

V. BleedingTooth exploit

앞서 설명한 BadChoice와 BadKarma를 사용하여 RCE 공격을 진행할 수 있다. 공격 단계는 아래 1,2,3...의 번호로 정리하였다. 1번부터 살펴보도록 하자.

I. BadKarma 우회

취약점을 이용하여 익스플로잇을 해야한다. ERTM 또는 스트리밍 모드일 때 l2cap_data_rcv가 호출된다. 이것이 호출되면 sk_filter로 인해 패닉 상태가 일어나기 때문에 A2MP 프로토콜 처리까지 도달할 수 없다. A2MP 레벨에서 일어나는 BadChoice를 악용하기 위해서는 먼저 BadKarma를 우회해야 한다(패닉을 피하기 위해)

L2CAP에 ERTM, Streaming 모드 말고도 코드를 살펴보면 UNACCEPT 모드가 존재한다. 해당 패킷을 보내면 서브루틴 l2cap_parse_conf_rsp를 호출한다. 여기서 일정 조건을 충족하면(패킷 보낼 때 설정하면 되는 간단한 if문이다), 원하는 모드로 변경할 수 있다. 위 방법을 이용하여 ERTM, Streaming 모드를 피해 A2MP 명령을 처리한다.

II. sk_filter() 조사

sk_filter 함수에서 sock 타입으로 받아야 하는 인수(sk)가 어떻게 이용되는지 살펴봐야 한다. 왜냐하면 이부분을 활용하여 포인터 역참조가 된다면 구조체 amp_mgr의 다른 member를 제어할 수도 있기 때문이다.

트리거에 사용할 멤버는 sock->sk_filter뿐이다(이것에 대해 조사하는 부분도 있는데 과감하게 넘어가겠다) Sk_filter는 sock에서 오프셋 0x110에 위치한다(그림1참고) Amp_mgr의 구조체 크기는 0x70바이트이기 때문에 단순히 amp_mgr만을 작성하여 sk_filter를 손상시키기는 어렵다. 다만, heap을 마음대로 구성할 수 있다면 sk_filter를 건드릴 수 있다. 커널이 직접 접근하는 메모리 영역은 slab이다.

III. heap primitive 찾기

heap을 구성하려면 당연히 동적 할당을 하는 primitive를 찾아야 한다. Primitive는 말그대로힙 할당을 도와주는 코드 조각을 찾아내는 것이다. primitive 조건은 아래와 같다.

a. 일정 크기의 메모리를 할당할 수 있다.

b. 공격자가 제어하는 데이터를 복사할 수 있다.

c. 해제할 때까지 메모리를 남겨둘 수 있어야 한다.

위 a, b를 충족하는 조건이 kmemdup 함수인데, A2MP_GETAMPASSOC_RSP 명령에 해당 함수가 존재한다. c조건의 경우 적절한 함수가 없어서 HCI 연결을 닫는 방법으로 진행했다(조금 원시적이고 시간이 느린다는 점은 감안해야 한다.)

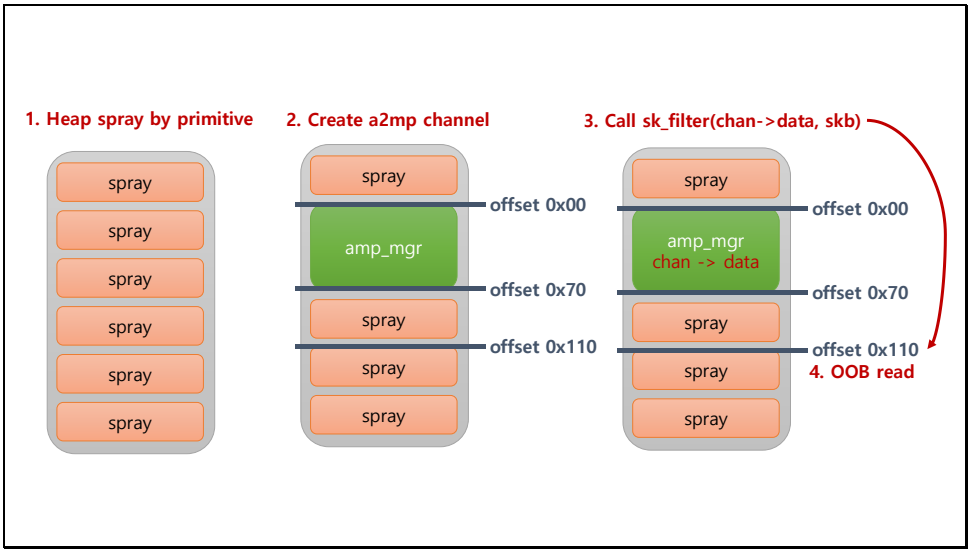
```
1. static int a2mp_getampassoc_rsp(struct amp_mgr *mgr, struct sk_buff *skb,
                                   struct a2mp_cmd *hdr)
2. {
3.       ...
4.       u16 len = le16_to_cpu(hdr->len);
5.       ...
6.       assoc_len = len - sizeof(*rsp);
7.       ...
```

```
 8.        ctrl = amp_ctrl_lookup(mgr, rsp->id);
 9.        if (ctrl) {
10.                  u8 *assoc;
11.
12.                  assoc = kmemdup(rsp->amp_assoc, assoc_len, GFP_KERNEL);
13.                  if (!assoc) {
14.                           amp_ctrl_put(ctrl);
15.                           return -ENOMEM;
16.                  }
17.
18.                  ctrl->assoc = assoc;
19.                  ctrl->assoc_len = assoc_len;
20.                  ctrl->assoc_rem_len = assoc_len;
21.                  ctrl->assoc_len_so_far = 0;
22.
23.                  amp_ctrl_put(ctrl);
24.        }
25.        ...
26. }
```

이 primitive를 이용하여 kmalloc-128 slab을 형성해줄 것이다. 굵기 처리한 부분을 살펴보면, kmemdup에서 복사가 가능하다. 위 함수로 kmemdup을 사용하기 위해서는 a2mp_getinfo_rsp를 이용해서 ctrl(변수)에 값을 넣어줘야 한다(중요하게 살펴볼 함수는 아니라서 코드는 첨부하지 않았다)

IV. OOB control

primitive로 해야할 것은 spray이다. Spray는 영단어 그대로 어떠한 공간에 같은 페이로드를 반복적으로 깔아서 말그대로 어떠한 것을 slab에 계속 써서 채우는 것이다(같은 내용을 분사하는 모양이라 spray라고들 한다) 128바이트 객체를 많이 할당하여 kmalloc-128 slab을 채우고, a2mp 채널을 새로 생성하여 amp_mgr 객체가 분사된 객체에 인접하여 오프셋 0x110의 값 제어가 가능하게 한다. 이후에 type confusion으로 역참조되면 OOB가 가능해진다.



V. Memory layout leak

이제 sk_filter가 역참조하는 포인터 제어가 가능해졌다. 그럼 이제 이 포인터를 어디를 가리켜야 할까? 해당 부분에서 BadChoice를 사용하여 포인터를 leak할 것이다. 아까 응답 우회해서

L2CAP_CONF_RSP를 전송하고 A2MP 채널을 ERTM 모드로 변경하려 하면 위에 있던 A2MP 서브 루틴이 실행되면서 포인터가 유출된다. 이때 l2cap_chan 객체 주소가 유출 될 수 있다.

l2cap_chan은 792바이트로, kmalloc-1024 slab에 할당된다. L2cap_chan의 주소를 유출하고 A2MP 채널을 해제하여 l2cap_chan도 해제한다. 다시 연결하여 kmalloc-1024 slab을 spray 한다. 이때 이전에 할당받은 l2cap_chan 객체 주소도 포함되어 있을 것이다(확률적인 부분이 있다. 익스플로잇에 있어서 중요한 계산은 아니므로 넘겼다)

VI. RIP control

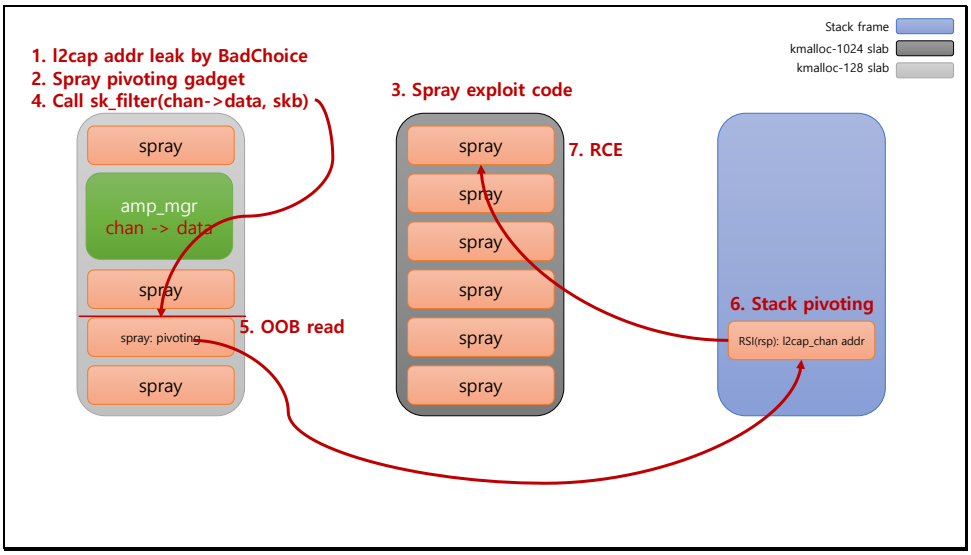
Sk_filter로 RIP를 제어할 수 있다. Sk_filter는 sk_filter_trim_cap 함수를 리턴한다. 여기서 사용하는 중요한 멤버들을 나열하면 아래와 같다.

```
sk->sk_filter->prog->bpf_func(skb, sk->sk_filter->prog->insnsi);
```

마지막 bpf_func의 두번째 인수 또한 sk->sk_filter에서 출발하므로 제어가 가능하다. 호출규약에 따라 두번째 인수는 RSI 레지스터에 담긴다.

VII. Kernel Stack pivoting

NX 보호기법 때문에 shellcode 직접 실행할 수 없다. ROP를 사용하기 위해서는 KASLR 우회를 해야한다. 때문에 RSP를 ROP 가젯이 있는 페이로드의 가짜 스택으로 리다이렉션하는 방향으로 잡았다. 6번 단락에서 RSI 레지스터도 우리가 제어할 수 있는 인자 중 하나라고 설명하였다. 이 RSI 값을 RSP로 옮기면 스택 피벗팅이 가능하다. 이때 사용한 가젯은 ROPgadget 툴을 이용하여 /boot/vmlinuz에서 추출하였다. 여기까지의 모든 과정을 끝내면 RCE를 달성할 수 있다.



Ⅵ. 닫는 글

실생활에서 쉽게 접할 수 있는 통신 기술이 블루투스라서 고른 주제이다. CVE 자체는 2020년이라 오래된 감이 있지만, 블루투스 통신 기술부터 heap 관련 익스플로잇 기술까지 폭넓게 공부할 수 있었다.

칼럼을 쓰면서 프로토콜과 커널 시스템 간의 유기성을 잘 연결하여 이해하기 쉽게 작성하려고 노력하였다. 해당 칼럼 독자를 커널 입문자로 생각하고 작성하다보니 slab 메모리 관리 등 cs 개념을 좀 더 자세히 싣고, 원글에서의 MTU 제한 등 까다로운 조건 해결 등은 과감히 넘어갔다. 디테일한 부분을 더 살펴보고 싶다면 참고문헌의 Bleedingtooth 원글을 찾아보는 것을 추천한다.

Ⅶ. 참고문헌

Bleedingtooth. (n.d.). Google Security-Research. https://github.com/google/security-research/tree/master/pocs/linux/bleedingtooth