Secrets behind PingPongRoot

Wen Xu @ Keen Team

HITB GSEC 2015
PingPongRoot

- Credit: wushi, Memeda Xu, James Fang and Leo.C
- First known 64bit root case in the world
- Universally applied bug and exploitation techniques in kernel
- Nominee for Pwnie Awards 2015
Roadmap

- CVE-2015-3636
- Kernel Exploit
- Future
CVE-2015-3636

Crash log from Trinity

A critical paging fault at 0x200200
struct sock sk:
PING socket object in kernel

Allocation: user_space_fd =
socket(AF_INET, SOCK_DGRAM, IPPROTO_ICMP);
LIST_POISON2 == 0x200200

ping_unhash(hlist_nulls_del) two times
0x200200 not mapped

Kernel crash due to a paging fault
Road to ping_unhash()

In user space: `connect()` on socket fd in user space with AF_UNSPEC
In kernel: `udp_disconnect()` on sk (kernel sock object)
First, to avoid crash: map 0x200200 in user space.

sock_put(sk) could then be invoked twice.

```c
34 void ping_unhash(struct sock *sk)
35 {
36     struct inet_sock *isk = inet_sk(sk);
37     pr_debug("ping_unhash(isk=%p, isk->num=%u)\n",
38             isk, isk->inet_num);
39     if (sk_hashed(sk)) {
40         write_lock_bh(&ping_table.lock);
41         hlist_nulls_del(&sk->sk_nulls_node);
42         sock_put(skb);
43         isk->inet_num = 0;
44         isk->inet_sport = 0;
45         sock_prot_inuse_add(sock_net(skb), skb->sk_prot, -1);
46         write_unlock_bh(&ping_table.lock);
47     }
48 }
49 EXPORT_SYMBOL_GPL(ping_unhash);
```
1. Allocate ping socket and get a file descriptor fd.
2. Map 0x200200 in user space.
3. Connect() to fd with sa_family == AF_INET
   To make sock sk hashed in kernel
4. Connect() to fd with sa_family == AF_UNSPEC twice
   sock_put(sk) is invoked twice
   ref_count of sk is to be 0 -> sk is freed in kernel space
5. We have a “dangling” file descriptor fd in our hand now.

```c
static inline void sock_put(struct sock *sk) {
    if (atomic_dec_and_test(&sk->sk_refcnt))
        sk_free(sk);
}
```
Exploitable?

If the space for the freed sk is re-controlled by us,

- In user space: simply `close(fd)`
- In kernel:
  - `inet_release()` invoked
  - As sk_prot is controlled by us, we can hijack the control flow of the kernel.
- Return to shellcode in usersland or ROP

```c
int inet_release(struct socket *sock)
{
    struct sock *sk = sock->sk;

    if (sk) {
        long timeout;

        [...] 

        if (sock_flag(sk, SOCK_LINGER) &&
            !(current->flags & PF_EXITING))
            timeout = sk->sk_lingertime;
        sk->sk = NULL;
        sk->sk_prot->close(sk, timeout);
    }

    return 0;
}
EXPORT_SYMBOL(inet_release);
```
Roadmap

- CVE-2015-3636
- Kernel Exploit
- Future
When it comes to UAF

- Most critical step: re-filling the freed vulnerable object
- This time, our target is struct sock object
- And it belongs to cache "PING",
  - kmem_cache_alloc("PING", priority &__GFP_ZERO);
- A custom-use cache
SLAB CACHE

A specific area for the allocation of kernel objects of particular type
Here we meet the type called “PING”
Challenges

1. Slab allocator
   - Natural separation between kernel objects
2. Few Candidate kernel objects
   - Most are not directly under the control of us
3. Multi-thread/core
   - Hard to achieve predictable memory layout
4. Controllable content
   - The content of most kernel objects are not totally under the control of us
What used to Re-Fill?
Candidate #1: kmalloc() buffers

- General use SLAB cache
- Rounded sizes
  - 32, 48, 64, 128, 256, 512, 1024...
- Easy to create: syscall sendmmsg()
- Size control: length of control message
- Content control: content of control message
Intuitive Idea

- Basically, a completely free slab has large probability to be recycled for future allocation.
- The fact kernel resumes memory provides us the opportunity to exploit UAF bugs anywhere.
  1. Fill slabs with totally PING socket objects
  2. Free all of them and spray kmalloc-x buffers
- Exactly possible, but ... out of control
- Low success rate in practical
SLUB Help Us?

- Newly adopted SLUB allocator tries to put the objects of the same size together, which de-separates the kernel objects to some extent.
- Then, does our target object have a size of 32, 48, 64, 128, 256 or 512?
  - Use kmalloc() buffers to re-occupy
  - Much more stable and accurate
- Limitations:
  - What if the size of the vulnerable object is 576, where $512 < 576 < 1024$?
  - Sizes of PING sock objects varies on different devices.
Memory Re-filling

Universal Solution #1
**RET2DIR**

- **ret2dir**: Rethinking Kernel Isolation (USENIX 14’)
  - Vasileios P. Kemerlis Michalis Polychronakis Angelos D. Keromytis
- **Physmap** is supposed to bypass kernel protections in the paper
  - SMEP, SMAp, PXN, PEN ...
- Would it help exploit kernel use-after-free bugs as well?
Physmap, the direct-mapped memory, is memory in the kernel which would directly map the memory in the user space into the kernel space.
The Return of Physmap

- Easy to create: iteratively calling mmap() in user space
  - `mmap((void *)addr, 0x10000000, PROT_READ | PROT_WRITE | PROT_EXEC, MAP_SHARED | MAP_FIXED | MAP_ANONYMOUS, -1, 0);`

- Data control: fully user-controlled (fill mmap()'ed area with our payload)

- Physmap with payload grows by occupying the free memory in the kernel
The Return of the Physmap

Size control:
- Physmap does not care about the size(type) of the vulnerable object.
- For its self, it has a large effective range.
- Large enough to fill any pages of freed memory theoretically.

---

Table 1: physmap characteristics across different architectures (x86, x86-64, AArch32, AArch64).

<table>
<thead>
<tr>
<th>Architecture</th>
<th>PHYS_OFFSET</th>
<th>Size</th>
<th>Prot.</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(3G/1G)</td>
<td>0xC00000000</td>
<td>891MB</td>
<td>RW</td>
</tr>
<tr>
<td>(2G/2G)</td>
<td>0x80000000</td>
<td>1915MB</td>
<td>RW</td>
</tr>
<tr>
<td>(1G/3G)</td>
<td>0x40000000</td>
<td>2939MB</td>
<td>RW</td>
</tr>
<tr>
<td>AArch32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(3G/1G)</td>
<td>0xC00000000</td>
<td>760MB</td>
<td>RWX</td>
</tr>
<tr>
<td>(2G/2G)</td>
<td>0x80000000</td>
<td>1784MB</td>
<td>RWX</td>
</tr>
<tr>
<td>(1G/3G)</td>
<td>0x40000000</td>
<td>2808MB</td>
<td>RWX</td>
</tr>
<tr>
<td>x86-64</td>
<td>0xFFFFFFFFF800000000000</td>
<td>64TB</td>
<td>RW(X)</td>
</tr>
<tr>
<td>AArch64</td>
<td>0xFFFFFFFFFC000000000000</td>
<td>256GB</td>
<td>RWX</td>
</tr>
</tbody>
</table>

Table [1] from ret2dir: Rethinking Kernel Isolation (USENIX 14’
Initial Plan

1. Allocate a large number of ping socket objects and then free all of them by triggering the bug.

2. Iteratively call mmap() in the user program and fill the area.

Hope the memory collision will happen?

- Low success rate
- Never let all the freed targeted vulnerable objects gather at one place
Reliability of Re-filling

Universal Solution #2
Reliable Memory Collision

- Goal: To make space for Physmap filled with our payload collide with PING sock objects in kernel
- Spray PING socket objects
  - In each step, every 500 PADDING PING objects normally release them by close()
  - 1 TARGET PING objects
    - Used to pwn and trigger the bug to release them
- That makes our vulnerable PING socket objects appear everywhere in kernel space
  - Scatter anywhere
- As long as one among these are overwritten, we win!
Information Leakage

Universal Solution #3
The data at a certain offset inside the object can be achieved by `ioctl()`.

Find an info leak to know whether our targeting PING socket object has already been covered by physemmap or not.
Notice: certain adjustment and optimization in practical root tool

Allocate hundreds of PING socket objects in group.

Every 500 padding objects with 1 targeting object considered as a vulnerable one.

Free padding PING socket objects normally by calling close()

Free targeting PING socket objects by triggering the bug

Such de-allocation generates large pieces of free memory (prepared for physmap)

Iteratively call mmap() in user space and fill the areas

Payload + magic number for re-filling checking

Iteratively call ioctl() on targeting PING socket objects

ioctl() returns magic number? Done.

Otherwise further physmap spraying is needed.

Summary
UNLEASH KERNEL UAF

- By leveraging physmap, we overcome all the challenges when exploiting such a UAF bug in kernel.
- In fact, it is a generic memory collision model in Linux kernel.
- Hard to mitigate due to kernel's inherent property.
Now we have full control of the content of a freed PING object with the corresponding dangling fd in our hand:

- In user space: simply close(fd)
- In kernel:
  - inet_release() invoked
  - sk_prot is overwritten to a prepared virtual address in user space
- Then PC value is under our control

```c
int inet_release(struct socket *sock)
{
    struct sock *sk = sock->sk;

    if (sk) {
        long timeout;

        [...]}

    if (sock_flag(sk, SOCK_LINGER) &&
        !(current->flags & PF_EXITING))
        timeout = sk->sk_lingertime;
    sock->sk = NULL;
    sk->sk_prot->close(sk, timeout);

    return 0;
}
EXPORT_SYMBOL(inet_release);
```
What does ShellCode do

- Leak kernel `sp` value (stack address)
- Thus we get address of `thread_info`
- Overwrite `addr_limit` of the current thread to 0x0
- Then we achieve kernel arbitrary read/write through pipe
What about 64bit devices?

- Bug existed? Yes.
- LIST_POISON2?
  - Still 0x200200 which can be mapped. Yes.
- Memory collision with phsymap? Yes.
- Return to shellcode in user space? No.
Bypassing PXN

- PXN prevents userland execution from kernel
- Return to physmap? Not executable on phones ;(
- ROP comes on stage
  - First step: leak kernel stack address
  - Second step: change addr_limit to 0
- Hardcoded addresses of gadgets ;(
In fact, we perform JOP

- Avoid stack pivoting in kernel which brings uncertainty
- Make full use of current values of the registers
  - X29 stores SP value on 64bit devices
- High 32bits of kernel addresses are the same
  - Only need to read/write low 32bits
- Work hard to find cool gadgets
Conclusion

- We successfully root most popular Android devices on market.
- Android version $\geq 4.3$
- First 64bit root case in the world (Samsung S6)
- Warranty safe
Roadmap

- CVE-2015-3636
- Kernel Exploit
- Future
64bit Devices could Be More Secure

- LIST_POISON2 in 64bit Android kernel
- 0x200200 Set as 0xDEAD000000000000
- Prevent memory collision with physmap
- To impose restrictions on memory resources for every user
- KASLR
- Days become harder for Linux kernel owners
- Where there is a will there is a way
ACKNOWLEDGEMENT

Thanks for contributions and inspirations

- Wushi
- JFang
- Leo.C
- Liang Chen
- Slipper
- Peter
Thank you!

Q & A