Own your Android! Yet Another Universal Root

Wen Xu[1] Yubin Fu[1]

[1]Keen Team
xuwen.sjtu@gmail.com qoobee1993@gmail.com

Abstract

In recent years, to find a universal root solution for Android becomes harder and harder due to rare vulnerabilities in the Linux kernel base and also the exploit mitigations applied on the devices by various vendors.

In this paper, we will present our universal root solution. The related vulnerability CVE-2015-3636, a typical use-after-free bug in Linux kernel is discussed in detail. Exploiting such a use-after-free in Linux kernel is truly difficult due to the separated allocation from the kernel allocator. We will show how we leverage this kernel use-after-free bug to achieve privilege promotion on most popular Android devices on market which have a version not less than 4.3, including the first 64bit root case in the world. In short, we will present a generic way to exploit use-after-free vulnerabilities in Linux kernel, which means one exploit applies to devices of all brands. All the current mitigations in the kernel like PXN are circumvented by this approach. And most importantly our unique and undocumented exploitation technique targeting kernel use-after-free bugs features stability and accuracy.

Bug analysis

The vulnerability is founded by Wen Xu and wushi of Keen Team with the help of our custom improved Trinity, which is a syscall fuzzer originally applied on PC Linux. We migrate it to ARM Linux for Android. The vulnerability is fixed in the latest Linux kernel source and assigned a CVE number, CVE-2015-3636.

The vulnerability lies in Linux kernel base part, which ensures that it can be used to do a general root. When one tries to call *connect* through a socket file descriptor, which is created by *socket(AF_INET, SOCK_DGRAM, IPPROTO_ICMP)* before, the kernel code shown in Figure 1 handles the user's request.

And if *sa_family* == *AF_UNSPEC* then the kernel calls the disconnect process specified by the protocol type. For a PING (ICMP) socket, the *disconnect* routine is shown in Figure 2.

We can see that the kernel finally calls *sk_prot_unhash(sk)*

```
1  int inet_dgram_connect(struct socket *sock,
2                         struct sockaddr * uaddr,
3                        int addr_len, int flags)
4  {
5      struct sock *sk = sock->sk;
6
7      if (addr_len < sizeof(uaddr->sa_family))
8          return -EINVAL;
9      if (uaddr->sa_family == AF_UNSPEC)
10         return sk->sk_prot->disconnect(sk, flags);
11
12     if (!inet_sk(sk)->inet_num && inet_autobind(sk))
13         return -EAGAIN;
14     [...]
15 }
16 EXPORT_SYMBOL(inet_dgram_connect);
```

Figure 1: inet_dgram_connect

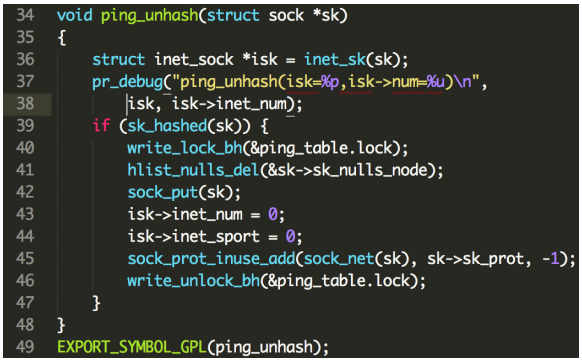
```
19 int udp_disconnect(struct sock *sk, int flags)
20 {
21     struct inet_sock *inet = inet_sk(sk);
22
23     sk->sk_state = TCP_CLOSE;
24     [...]
25     if (!(sk->sk_userlocks & SOCK_BINDPORT_LOCK)) {
26         sk->sk_prot->unhash(sk);
27         inet->inet_sport = 0;
28     }
29     sk_dst_reset(sk);
30     return 0;
31 }
32 EXPORT_SYMBOL(udp_disconnect);
```

Figure 2: udp_connect

which is *ping_unhash()* in Figure 3 for a PING (ICMP) socket.

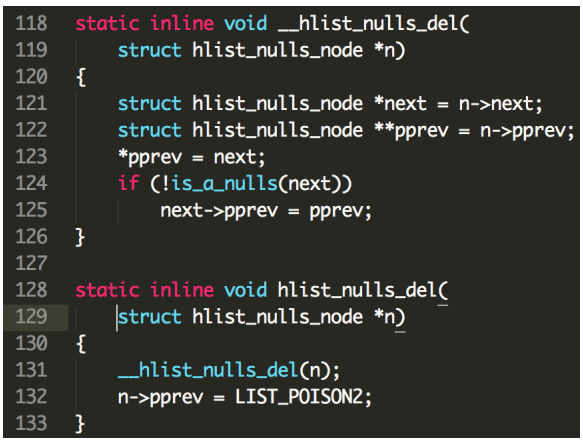
As shown in the source, if sock object (ICMP socket) *sk* is hashed, then it will try to delete its *sk_nulls_node* stored in a hlist in the kernel (Figure 4).

We can figure out that after *n* (*sk->sk_nulls_node*) being deleted, the value of *n->pprev* becomes *LIST_POISON2*, which is defined as a constant value. Practically it is $0x200200$ both in Android 32bit kernel and 64bit kernel.



```
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(sk);
43          isk->inet_num = 0;
44          isk->inet_sport = 0;
45          sock_prot_inuse_add(sock_net(sk), sk->sk_prot, -1);
46          write_unlock_bh(&ping_table.lock);
47      }
48  }
49  EXPORT_SYMBOL_GPL(ping_unhash);
```

Figure 3: ping_unhash



```
118  static inline void __hlist_nulls_del(
119      struct hlist_nulls_node *n)
120  {
121      struct hlist_nulls_node *next = n->next;
122      struct hlist_nulls_node **pprev = n->pprev;
123      *pprev = next;
124      if (!is_a_nulls(next))
125          next->pprev = pprev;
126  }
127
128  static inline void hlist_nulls_del(
129      struct hlist_nulls_node *n)
130  {
131      __hlist_nulls_del(n);
132      n->pprev = LIST_POISON2;
133  }
```

Figure 4: hlist_nulls_del

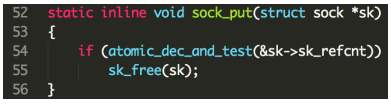
This virtual address can be mapped by the attacker in the user space.

However something amazing happens when calling *connect* a second time. After the socket object is deleted from *hlist*, it still remains hashed since whether it is hashed or not depends on *sk->sk_node* which is not changed during the first connection.

Hence the kernel goes into that *if* branch, and then tries to delete *sk->sk_nulls_node* again. When the kernel executes **pprev = next*, it will crash because the current value of *pprev* is $0x200200$, and if this virtual address is not mapped in the user space, then a critical page fault will happen in the kernel which leads to a panic. $0x200200$ should be mapped in the user space before the second ICMP socket connection to avoid crashes. However, such a local DoS is not the whole story of this vulnerability.

Taking a brief look at the codes (Figure 5) after *hlist_nulls_del* is called, one could find that *sock_put(sk)* is really suspicious.

Every time the kernel goes into that *if* branch, it minuses the *reference count* of the sock object in the kernel by one. And most importantly, it will check whether the *reference*



```
52  static inline void sock_put(struct sock *sk)
53  {
54      if (atomic_dec_and_test(&sk->sk_refcnt))
55          sk_free(sk);
56  }
```

Figure 5: sock_put

count becomes zero or not. If it is zero, the sock object is going to be freed. That means if one tries to connect such a sock object for a second time, the *reference count* of it will come to zero, thus the kernel will free it. But the file descriptor handled in the user program is still related with the corresponding sock object in the kernel, which is a typical use-after-free vulnerability.

Proof-of-Concept

Here is a piece of PoC of CVE-2015-3636:

```
int sockfd = socket(AF_INET,
    SOCK_DGRAM, IPPROTO_ICMP);
struct sockaddr addr
    = { .sa_family = AF_INET };
int ret = connect(sockfd, &addr,
    sizeof(addr));
struct sockaddr _addr
    = { .sa_family = AF_UNSPEC };
ret = connect(sockfd, &_addr, sizeof(_addr));
ret = connect(sockfd, &_addr, sizeof(_addr));
```

connect must be called first with a *AF_INET sa_family* to make *sk* (that vulnerable sock object) hashed in the kernel, otherwise that *if* cannot be reached at all.

Notice that this PoC only takes effect on Android devices. The range of the group id which is permitted to create a PING socket is set in */proc/sys/net/ipv4/ping_group_range*. On Android devices, a common user has the privilege to create PING sockets by default while on PC Linux, normally nobody has the privilege to create them.

Exploitation

Goal

Till now we have figured out this typical use-after-free vulnerability and a dangling file descriptor in the user space pointing to the PING sock object in the kernel can be achieved by the attacker. Next, we will present our approach to overwrite the sock object and call something to reuse the PING sock object. After that we are able to execute arbitrary code in the kernel and finally achieve privilege escalation on the Android device.

In practical, we use the *close* function of the socket object. When *close(sockfd)* is called, the kernel finally goes into the codes shown in Figure 6.

The kernel calls *inet_release* to release the socket object in the kernel related with the *sockfd*. And at the bottom of this function, it calls *sk->sk_prot->close()*.

In fact, *sk_prot* is a member of struct *sk* type, which points to a certain amount of function pointers. What function

```
 99  int inet_release(struct socket *sock)
100  {
101      struct sock *sk = sock->sk;
102
103      if (sk) {
104          long timeout;
105
106          [...]
107
108          if (sock_flag(sk, SOCK_LINGER) &&
109              !(current->flags & PF_EXITING))
110              timeout = sk->sk_lingertime;
111          sock->sk = NULL;
112          sk->sk_prot->close(sk, timeout);
113      }
114      return 0;
115  }
116  EXPORT_SYMBOL(inet_release);
```

Figure 6: inet_release

pointers that *sk* has depends on its protocol type, including TCP, UDP, PING and so on.

If the content of the freed PING sock object *sk* is refilled by the data which is fully controlled by us, then *sk->sk_prot* is surly under our control. It can be specified as a virtual address in the user space. Then the address of function pointer *sk->sk_prot->close* is under our control. If *PAN* (Privileged Access Never) is not applied in the kernel, then we can finally control the PC register in the kernel context. As a matter of fact, *PAN* (Privileged Access Never) has not been adopted by any current Android device on market, so here we do not take it into consideration.

Generally speaking, two important factors are considered in our solution. One factor is that the vulnerable sock object should be refilled stably and reliably and the other one is that the re-filling content should be fully controlled by us.

Re-filling

The most difficult thing in this root exploit is to overwrite the freed sock object with the proper data we want, and also we should ensure that the whole process is stable and reliable due to the user experience of a successful root tool for those Android users. And for sure the unique and undocumented exploitation technique we used to do a reliable and accurate re-filling of the freed object is the highlight of our work.

Considering the heap management mechanism adopted by the Linux kernel, it currently takes the SLAB/SLUB allocator for the efficient allocation of the kernel object [3].

Different SLABs created for different objects in the kernel, and there is no doubt that a PING cache is created for our vulnerable PING sock objects. Such separations can also be widely seen in user programs, like *Isolated Heap* for Internet Explorer on Windows and *PartitionAlloc* for Chrome, etc. When facing this in the kernel, it is not easy for the attacker to use the object of type **A** to occupy an area which was once for the object of a different type **B**.

Another factor which brings uncertainty comes from the multi-threading support by the Linux kernel. It is a common scene that hundreds of tasks run on a single system (Android device) concurrently, and the execution of each task may cause the allocation or de-allocation of objects in the kernel, which will finally influence the kernel heap layout. However, a predictable heap layout is so important in a use-after-free exploit as known.

Things are not too bad if the system supports the SLUB allocator, which is true for most Android devices on market. If the vulnerable object like PING sock has a size of 512 or 1024, then re-filling is much easier. Because the SLUB allocator tends to put the objects of the same size into one SLAB cache, which means that if the vulnerable objects size is 512, then it will be placed into one SLAB with some other *kmalloc-512* objects. Thus the content of the freed vulnerable object under this circumstance can be fully controlled by leveraging *malloc-512* objects. In fact, the *kmalloc-512* object is able to be created in the user program. One way is to use *sendmmsg*. During the execution of *sendmmsg*, the kernel will use *kmalloc* to allocate a buffer in the kernel to store our transfer data packet temporarily. The size of the transfer data can be specified by us and it should be set to 512 in this situation. And the content of the buffer can also be fully controlled by us since it is just the data we want to transfer through *sendmmsg*. The whole process is shown in Figure 7.

Figure 7: Content control by sendmmsg

Notice that this solution is an excellent one for re-filling use-after-free objects in the kernel, but however it has a serious limitation: the vulnerable object should have a size which a common-use SLAB has. In other words, it must have a size which can be allocated by kmalloc in the kernel. For example on some Android devices, the PING sock object has a size of 576, which is between 512 and 1024. Then the solution above is no longer valid.

Theoretically it is possible to use *kmalloc-size* objects to re-fill any other objects in the kernel. And it takes advantage of the principal that when a whole SLAB is free then it may be recycled by the kernel for future use. Based on this, we can first spray our PING sock objects to occupy some SLABs where no other kernel objects are stored. After that we try to free all of them by triggering the use-after-free vulnerability. Several completely free SLABs are generated. Then we allocate a certain amount of transfer buffers through *sendmmsg*, which all have a size of 512. Chances are that these buffers occupy the SLABs once storing PING

sock objects and the re-filling is done (Figure 8). However, this approach is very hard to control and has huge uncertainty. We do not exactly know which transfer buffer has been stored in the SLAB where PING sock objects were stored before, and that makes the whole root exploit not stable and reliable.

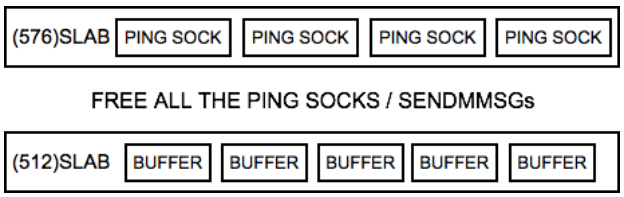


Figure 8: Mis-alignment

Furthermore, the sizes of PING sock objects on different devices are diverse. If we need a generic applied solution to all the Android devices, then it should not rely on the sizes of PING sock objects on these devices.

Here our tricky technique to achieve a stable re-filling on the freed PING sock objects to exploit this vulnerability will be presented. Notice that we do not care about the sizes of PING sock objects on the device. This time, we do not even use other kernel objects to do the re-filling work, but use *physmap* instead.

The *physmap* in the kernel is once mentioned in [1]. It is a large piece of memory in the kernel space and directly maps the memory in the user space into the kernel space for promoting the performance of the system.

That means we can spray data in the user space by repeatedly calling *mmap* and much of them will be directly appear in the kernel space. Our intention is to use these user space data to cover the freed PING sock object and several issues are needed to be concerned about.

As show in Figure 9, we can see that in kernel space, the *physmap* and the SLABs are normally located at different places. The *physmap* begins at a relatively high location, while the SLABs usually locate at a relatively low location. In order to make them collide in the middle of the kernel space, the first step we should do is to spray some other kernel objects to push the kernel allocator to begin allocating kernel objects at a higher place, which improves the possibility of the memory reusing between the SLABs and the *physmap*. This step is called *lifting*. In our root exploit targeting CVE-2015-3636, we just take the PING sock object itself for lifting, since it is easy to allocate (by calling *socket*) and de-allocate (by calling *close*) in the kernel.

After *lifting* (Figure 10), a certain amount of PING sock objects are allocated, but this time they are seen as the vulnerable ones. Due to the previous lifting, these PING sock objects will be located at a higher place in the kernel space.

During the later de-allocation, we normally free these PING sock objects for *lifting* but for these targeting vulnerable ones, we release them by triggering the vulnerability which means making two connections through one PING socket.

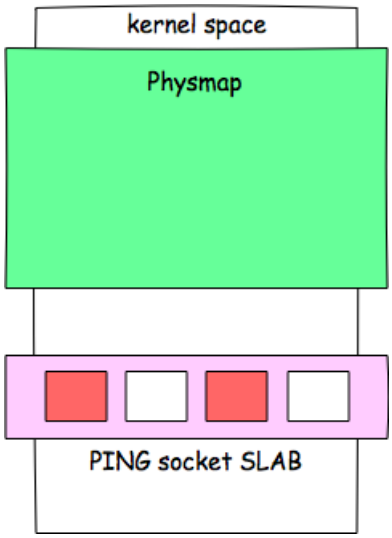


Figure 9: Physmap

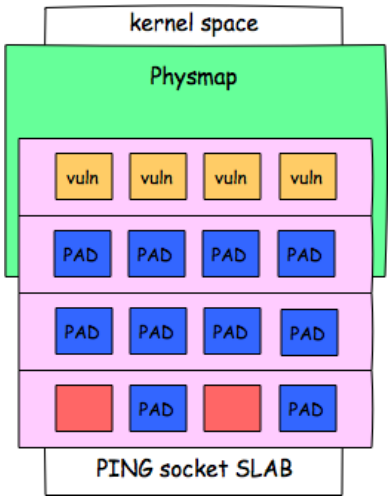


Figure 10: Memory collision with Physmap after lifting up

Then we repeatedly call *mmap* and fill the mapped area in the user space with the data we want. The spraying data is grouped in 8 dwords. Every 8 dwords is the same.

One big problem is that how we can know it is the time to stop spraying when our targeting vulnerable PING sock objects have already been covered by the data in the *physmap*. To solve this issue, among our 8 dwords, besides some key values to control the flow and avoid kernel crashes on the road to the final control, the specific entry is filled with the predefined magic value.

Every time we finish spraying a certain amount of data,

we call *ioctl(sockfd, SIOCGSTAMPNS, (struct timespec*))* on these targeting vulnerable PING sock objects.

```
79  int sock_get_timestampns(struct sock *sk,
80      struct timespec __user *userstamp)
81  {
82      struct timespec ts;
83      if (!sock_flag(sk, SOCK_TIMESTAMP))
84          sock_enable_timestamp(sk, SOCK_TIMESTAMP);
85      ts = ktime_to_timespec(sk->sk_stamp);
86      if (ts.tv_sec == -1)
87          return -ENOENT;
88      if (ts.tv_sec == 0) {
89          sk->sk_stamp = ktime_get_real();
90          ts = ktime_to_timespec(sk->sk_stamp);
91      }
92      return copy_to_user(userstamp, &ts,
93          sizeof(ts)) ? -EFAULT : 0;
94  }
95  EXPORT_SYMBOL(sock_get_timestampns);
```

Figure 11: Information leak

As seen in Figure 11, it reads out the member *sk_stamp* of *sk*. Thus by calling *ioctl* with the specific arguments, we can successfully leak out a dword value at a fixed offset inside the PING sock object. We can compare this value with the magic value we previously filled in, and then get to know whether the covering is done or not. This step ensures the reliability of our root exploit.

When we have already got the exactly covered PING sock object, we call *close* on the dangling file descriptor of it. As described above, the kernel will eventually call *sk->sk_prot->close* and now *sk_prot* is already under our control, and its address is pointing to a fully controlled space in the user land, thus the function pointer of *close* is controlled. And finally we control the value of the PC register in the kernel context.

Notice that our final re-filling solution does not depend on any specific configuration or details of kernels on these Android devices, it just leverages an inherent directly mapped area in the Linux kernel space to achieve our goal.

64bit devices

Our root exploitation mentioned above is also applied to Android 64bit devices. There are mainly two reasons:

A. The value of *LIST_POISON2* on Android 64bit devices remains to be $0x200200$, which is a mappable virtual address in the user space. For PC Linux 64bit, this value is $0xdead000000000000$, which has 64bit size long and definitely cannot be mapped since it is even out of the range of the virtual address space on 64bit system. And if this value cannot be mapped, then the crash cannot be avoided when we *connect* to the PING socket a second time.

B. The *physmap* is proved to be able to cover SLABs on 64bit devices in practical. In fact, the *physmap* in 64bit kernel maps all the user space memory into the kernel so the spraying also takes effect [1].

And eventually we manage to control the PC register in the kernel of Android 64bit devices by the approach described above.

ROOT

After we control the PC register in the kernel context, we plan to do code execution and achieve root privilege on the devices, which is always our final goal. The basic approach is to rewrite the *addr_limit* of current task to 0 and thus get arbitrary read/write in the kernel space [2]. After that we rewrite some credential structures in the kernel to promote our privilege.

For these devices which do not have *PXN* on, things are very easy. We just set the *close* function pointer to a virtual address in the user space, and place a piece of *shellcode* there, which is used to change the *addr_limit* of the current task to 0.

For these devices which have *PXN* on, *ret2usr* attack [4] is no longer effective. Somehow we have to conduct ROP [5] to achieve our goal. However in order to design a stable exploit, we use kernel JOP (Jump-Oriented Programming) [6] to rewrite the *addr_limit* of the current task to 0 instead of ROP.

1) Referred registers during JOP

During JOP, many registers may change their original values. In fact, their original values are probably lost if we do not care about them. In our root exploit, we call *close* on the dangling socket file descriptor and thus go into the JOP chain placed in the user land. During the whole execution of our JOP chain, we only corrupt the values of *r0*, *r1*, *r2*, *r3*, *r4* and *r5*. Changing the value of any other register will lead to unexpected kernel crashes later. Some key registers values are required to remain the same.

2) Keeping the value of *SP*

The main reason we choose JOP instead ROP is that in ROP usually we need to do stack pivot, which means pivoting the kernel stack into the user space, and such behaviors bring uncertainty since the *SP* value is corrupted for a while during the exploit. Register *SP* is truly critical during the execution of the whole kernel and it is not an advisable choice to corrupt its value at anytime.

3) Avoiding data corruption on the stack

We should avoid such gadgets during JOP like

```
str x2, [sp, 0x20] # 32bit
str x2, [x29, 0x20] # 64bit
```

Register *x29* usually stores the *SP* value in 64bit Linux kernel. These gadgets corrupt the data on the stack, and thus influence the future execution flow of the kernel. Such behavior also brings uncertainty and should be avoided.

4) Exploring core gadgets

Our JOP chain mainly has two tasks. The first task is to leak out the *SP* value from the kernel, and then we can get the address of *task_struct* of the current task from the *SP* value [2]. The second task is to rewrite the *addr_limit* of the current *task_struct*, which leads to an arbitrary read/write in

the kernel space. The core gadget we try to find out looks like:

```
str x1, [x0, 0x14]
ldr x1, [x2, 0x10]
blr x1
```

Take this as an example. For leaking, the value of register *x0* should be a virtual address in the user space, then we can read out the value of register *x1*, which should be the kernel *SP* value. And we can return back from JOP by jumping to the correct return address pointed by register *x2*, whose value should also be an address in the user space.

For rewriting, the value of register *x1* is 0, and the value of *x0* should be an address related with the address of *addr_limit*. And then we return back to the original return address.

Totally there are two steps, leaking and rewriting. And we try to find out such gadgets in the boot image of various devices which have *PXN* on based on the rules described above.

5) Leaking tricks

A. On 64bit Android devices, register *x29* stores the value of *SP* of the kernel as usual. Thus such instructions can be used to get the kernel stack address:

```
mov x0, sp # 32bit
mov x0, x29 # 64bit
```

B. For the 64bit devices, the high 32 bits of a kernel virtual address remain the same. So to leak out the low 32 bits of the kernel stack address is enough for our attack.

```
str w2, [x4, 0x80]
```

6) Rewriting tricks

When it comes to some ROMs of these Android devices on market, there exists a gadget in the image and we can leverage it to achieve arbitrary address write in the kernel, then things are done.

Otherwise we have two choices.

A. Direct way

```
mov x1, 0
str x1, [addr_limit]
```

B. Indirect way

```
mov x1, [user_space_address]
str x1, [addr_limit]
```

And sometimes we can also use

```
str w1, [addr_limit]
```

and write two times to set *addr_limit* to 0.

Conclusion

In this paper, we present the details of CVE-2015-3636 and how Keen Team leverages it to achieve universal root on most popular Android devices (version>=4.3) on market. We apply our exploitation techniques to root the 64bit devices, which is the first case in the world as known. In addition, through certain tricks about applying JOP in the Android kernel, PXN can be fully bypassed reliably.

Acknowledgement

Thanks to wushi, the leader of Keen Team, for his great contribution to CVE-2015-3636. And also thanks to James Fang of Keen Team for his great contribution to the development of our root solution. Thanks to Liang Chen, Siji Feng and Peter Hlavaty of Keen Team for their inspirations on the idea of this generic root exploit.

References

1. V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis. ret2dir: Rethinking kernel isolation. USENIX Security Symposium, 2014.

2. Jon Oberheide, Dan Rosenberg. Stackjacking Your Way to grsecurity/PaX Bypass. INFILTRATE 2011.

3. https://www.kernel.org/doc/Documentation/vm/slub.txt.

4. Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. kGuard: Lightweight Kernel Protection against Return-to-user Attacks. USENIX Security Symposium, 2012.

5. Marco Prandini and Marco Ramilli. Return-oriented programming. Security and Privacy, IEEE, 2012.

6. Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, Zhenkai Liang. Jump-Oriented Programming: A New Class of Code-Reuse Attack. Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security. ACM, 2011.