Exploiting Uses of Uninitialized Stack Variables in Linux Kernels to Leak Kernel Pointers

Haehyun Cho, Jinbum Park, Joonwon Kang, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, Gail-Joon Ahn
Uninitialized variables in the stack

```c
void func () {
    int num;
    int ret;
    ...
}
```
void func () {
    int num;
    int ret;
    ...
}

Uninitialized variables in the stack

Kernel Stack

0x 1111 2222 3333 4444
0x aaaa bbbb cccc dddd
... ... ... ...
... ... ... ...
... ... ... ...

Base – 8n
Uninitialized variables in the stack

```c
void func () {
    int num = 0;
    int ret = 0;
    struct data_struct = {0,};
    ...
}
```
Unexpected information leaks

```c
void func () {
    int num = 0;
    int ret = 0;
    struct data_struct = {0,};
    ...
}
```

- If uninitialized data can be copied to the user-space...

![Kernel Stack Diagram]

```
<table>
<thead>
<tr>
<th></th>
<th>0x 0000 0000 0000 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x 0000 0000 0000 0000</td>
</tr>
<tr>
<td></td>
<td>0x 0000 0000 0000 0000</td>
</tr>
<tr>
<td></td>
<td>0x 0000 0000 0000 0000</td>
</tr>
<tr>
<td></td>
<td>0x 0000 0000 0000 0000</td>
</tr>
<tr>
<td></td>
<td>0x 0000 0000 0000 0000</td>
</tr>
<tr>
<td></td>
<td>0x 0000 0000 0000 0000</td>
</tr>
<tr>
<td></td>
<td>0x 0000 0000 0000 0000</td>
</tr>
</tbody>
</table>
```

SP ➔ Base – 8n
Real-world example (CVE-2016-4486)

/* file: net/core/rtnetlink.c */
static int rtnl_fill_link_ifmap(struct sk_buff *skb, struct net_device *dev)
{
    // all fields in the map object are initialized
    struct rtnl_link_ifmap map = {
        .mem_start = dev->mem_start,
        .mem_end = dev->mem_end,
        .base_addr = dev->base_addr,
        .irq = dev->irq,
        .dma = dev->dma,
        .port = dev->if_port,
    };

    // kernel data leak to the user-space
    if(nla_put(skb, IFLA_MAP, sizeof(map), &map))
        return -EMSGSIZE;
    return 0;
}
Real-world example (CVE-2016-4486)

/* file: net/core/rtnetlink.c */
static int rtnl_fill_link_ifmap(struct sk_buff *skb, struct net_device *dev) {
  //all fields in the map object are initialized
  struct rtnl_link_ifmap map = {
    .mem_start = dev->mem_start,
    .mem_end = dev->mem_end,
    .base_addr = dev->base_addr,
    .irq = dev->irq,
    .dma = dev->dma,
    .port = dev->if_port,
  };

  //kernel data leak to the user-space
  if(nla_put(skb, IFLA_MAP, sizeof(map), &map))
    return -EMSGSIZE;

  return 0;
}
Basic security principle of the OS kernels

- Applications are not allowed to access the kernel memory
- Restricted Kernel data must not leave the kernel memory
Information leaks are not rare

In Linux kernel,

• Information leak vulnerabilities are the most prevalent type [1].
• Kernel Memory Sanitizer (KMSAN) discovered more than a hundred uninitialized data use bugs [2].

---

Information leaks are not rare

In Linux kernel,

• Information leak vulnerabilities are the most prevalent type [1].
• Kernel Memory Sanitizer (KMSAN) discovered more than a hundred uninitialized data use bugs [2].

However, these vulnerabilities are commonly believed to be of low risks [3]. → not assigned any CVE entries and not patched in some cases

---

Survey result on information leak CVEs

The number of information leak CVEs related to uses of uninitialized data between 2010 and 2019.

<table>
<thead>
<tr>
<th># of CVEs</th>
<th>Total</th>
<th>Stack-base</th>
<th>Heap-base</th>
<th># of exploits</th>
</tr>
</thead>
<tbody>
<tr>
<td>87</td>
<td>76</td>
<td>11</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

• The majority of these CVEs are stack-based information leaks.

• 0 public exploit and 0 proof-of-vulnerability (PoV)
  • Even with a PoV, it is difficult to evaluate the exploitability

• Only once CVE (CVE-2017-1000410) mentions that “Potential of leaking kernel pointers and bypassing KASLR”
Our Goal

• Reveal the actual exploitability and severity of information leak bugs

• Converts stack-based information leaks in Linux kernels into vulnerabilities that leak kernel pointer values.
  • We focus on leaking pointer values that are pointing to (1) kernel functions or (2) the kernel stack.
Challenges in Exploitation

- Computing the offset to uninitialized data from the kernel stack base.

![Kernel Stack Diagram]
Challenges in Exploitation

• Computing the offset to uninitialized data from the kernel stack base.

• Storing kernel pointer values at a leak offset.
Challenges in Exploitation

• Computing the offset to uninitialized data from the kernel stack base.

• Storing kernel pointer values at a leak offset.

• Handling data leaks that are less than 8 bytes.
Our Approach

PoVs → Analysis → Exploitability and exploits
Computing the Leak Offset

Stack footprinting
1. Fill the kernel stack
Computing the Leak Offset

Stack footprinting
1. Fill the kernel stack
2. Trigger a vulnerability
3. Check the footprint

<table>
<thead>
<tr>
<th>Kernel Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x ????? ????? ????? ?????</td>
</tr>
<tr>
<td>0x ????? ????? ????? ?????</td>
</tr>
<tr>
<td>0x ????? ????? ????? ?????</td>
</tr>
<tr>
<td>0x ????? ????? ????? ?????</td>
</tr>
<tr>
<td>0x ????? ????? ????? ?????</td>
</tr>
<tr>
<td>0x ????? ????? ????? ?????</td>
</tr>
<tr>
<td>0x ????? ????? ????? ?????</td>
</tr>
<tr>
<td>0x ????? ????? ????? ?????</td>
</tr>
<tr>
<td>0x ????? ????? ????? ?????</td>
</tr>
<tr>
<td>0x ????? ????? ????? ?????</td>
</tr>
<tr>
<td>... ... ... ... ...</td>
</tr>
</tbody>
</table>

Base – 8n
Computing the Leak Offset

Stack footprinting
1. Fill the kernel stack
2. Trigger a vulnerability
3. Check the footprint
   ![Kernel Stack Table]

4. Compute the offset
   → Leak offset = Base - 24

![Kernel Stack Table]
Extensive Syscall Testing with the LTP

• Linux Test Project (LTP) provides concrete test cases for system calls.

• Three additional steps onto each syscall test case
  1. Spraying the kernel stack with a magic value
  2. Finding kernel pointer values stored in the stack
  3. Recording context information
Syscall Testing with the LTP

1. Fill the kernel stack

<table>
<thead>
<tr>
<th>Kernel Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x 1122 3344 5566 7788</td>
</tr>
<tr>
<td>0x 1122 3344 5566 7788</td>
</tr>
<tr>
<td>0x 1122 3344 5566 7788</td>
</tr>
<tr>
<td>0x 1122 3344 5566 7788</td>
</tr>
<tr>
<td>0x 1122 3344 5566 7788</td>
</tr>
<tr>
<td>0x 1122 3344 5566 7788</td>
</tr>
<tr>
<td>0x 1122 3344 5566 7788</td>
</tr>
<tr>
<td>0x 1122 3344 5566 7788</td>
</tr>
<tr>
<td>0x 1122 3344 5566 7788</td>
</tr>
<tr>
<td>... ... ... ...</td>
</tr>
</tbody>
</table>

Base – 8n
Syscall Testing with the LTP

1. Fill the kernel stack
2. Execute a syscall using a testcase
3. Inspect the kernel stack

<table>
<thead>
<tr>
<th>Kernel Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x ???</td>
</tr>
<tr>
<td>0x ???</td>
</tr>
<tr>
<td>0x ???</td>
</tr>
<tr>
<td>0x ???</td>
</tr>
<tr>
<td>0x ???</td>
</tr>
<tr>
<td>0x ???</td>
</tr>
</tbody>
</table>

Kernel pointer to a kernel function

<table>
<thead>
<tr>
<th>Kernel pointer to the kernel stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x ???</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

Base – 8n
Syscall Testing with the LTP

1. Fill the kernel stack
2. Execute a syscall using a testcase
3. Inspect the kernel stack
4. Record the context information

<table>
<thead>
<tr>
<th>Offset</th>
<th>Base - 24</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Kernel code</td>
</tr>
<tr>
<td>Syscall</td>
<td>mmap</td>
</tr>
<tr>
<td>Args</td>
<td>0,8,0,0,-1,0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset</th>
<th>Base - 64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Kernel stack</td>
</tr>
<tr>
<td>Syscall</td>
<td>mmap</td>
</tr>
<tr>
<td>Args</td>
<td>0,8,0,0,-1,0</td>
</tr>
</tbody>
</table>
Stack Spraying via BPF

• The extended Berkeley Packet Filter (BPF) allows users to make a program and execute it inside the kernel.

• BPF program has its own stack (512 bytes)

<table>
<thead>
<tr>
<th>Kernel Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x ????? ???? ???? ????</td>
</tr>
<tr>
<td>0x ????? ???? ???? ????</td>
</tr>
<tr>
<td>0x ????? ???? ???? ????</td>
</tr>
</tbody>
</table>

Spraying the frame pointer by crafting BPF instructions

Pointer to the base address of a BPF program
Stack Spraying via BPF

• The extended Berkeley Packet Filter (BPF) allows users to make a program and execute it inside the kernel.

• BPF program has its own stack (512 bytes)

• If we leak the frame pointer, we can identify the layout of kernel stack
Handling Small Data Leaks

• Need the most important 52 bits (7 bytes) of a kernel stack address
  • the kernel stack is aligned by the size of a page (i.e., 4KB, by default)

• If we only know 4 bytes ... ?
  → Guess and check!

| 0x ffff | ff04 | 2000 | 0000 |
Handling Small Data Leaks

• Need the most important 52 bits (7 bytes) of a kernel stack address
  • the kernel stack is aligned by the size of a page (i.e., 4KB, by default)

• If we only know 4 bytes ... ?
  → Guess and check!

• e.g., spraying (FP — 0x0000 0000 3000 0000)

```
0x ffff ff03 ???? ????  
```

Hidden data < 0x3000 0000
Handling Small Data Leaks

• Need the most important 52 bits (7 bytes) of a kernel stack address
  • the kernel stack is aligned by the size of a page (i.e., 4KB, by default)

• If we only know 4 bytes ... ?
  → Guess and check!

• e.g.,
  spraying (FP — 0x0000 0000 3000 0000)  \( 0x \ ffff \ ff03 \ ???? \ ???? \)
  Hidden data < 0x 3000 0000

  spraying (FP — 0x0000 0000 1234 0000)  \( 0x \ ffff \ ff04 \ ???? \ ???? \)
  Hidden data > 0x 1234 0000
Evaluation
Finding pointers with the LTP framework
## Summary of exploitation results

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Leak Size</th>
<th>CVSS</th>
<th>Exploitation Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2018-11580</td>
<td>4 bytes</td>
<td>2.1</td>
<td>Bypass KASLR</td>
</tr>
<tr>
<td>CVE-2016-4569</td>
<td>4 bytes</td>
<td>2.1</td>
<td>Bypass KASLR</td>
</tr>
<tr>
<td>Fixes: 372f525</td>
<td>4 bytes</td>
<td>N/A</td>
<td>Bypass KASLR</td>
</tr>
<tr>
<td>CVE-2016-4486</td>
<td>4 bytes</td>
<td>2.1</td>
<td>Reveal the kernel stack base</td>
</tr>
<tr>
<td>CVE-2016-5244</td>
<td>1 byte</td>
<td>5</td>
<td>Failed</td>
</tr>
</tbody>
</table>
Summary of exploitation results

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Leak Size</th>
<th>CVSS</th>
<th>Exploitation Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2018-11580</td>
<td>4 bytes</td>
<td>2.1</td>
<td>Bypass KASLR</td>
</tr>
<tr>
<td>CVE-2016-4569</td>
<td>4 bytes</td>
<td>2.1</td>
<td>Bypass KASLR</td>
</tr>
<tr>
<td>Fixes: 372f525</td>
<td>4 bytes</td>
<td>N/A</td>
<td>Bypass KASLR</td>
</tr>
<tr>
<td>CVE-2016-4486</td>
<td>4 bytes</td>
<td>2.1</td>
<td>Reveal the kernel stack base</td>
</tr>
<tr>
<td>CVE-2016-5244</td>
<td>1 byte</td>
<td>5</td>
<td>Failed</td>
</tr>
</tbody>
</table>
### Summary of exploitation results

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Leak Size</th>
<th>CVSS</th>
<th>Exploitation Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2018-11580</td>
<td>4 bytes</td>
<td>2.1</td>
<td>Bypass KASLR</td>
</tr>
<tr>
<td>CVE-2016-4569</td>
<td>4 bytes</td>
<td>2.1</td>
<td>Bypass KASLR</td>
</tr>
<tr>
<td>Fixes: 372f525</td>
<td>4 bytes</td>
<td>N/A</td>
<td>Bypass KASLR</td>
</tr>
<tr>
<td>CVE-2016-4486</td>
<td>4 bytes</td>
<td>2.1</td>
<td>Reveal the kernel stack base</td>
</tr>
<tr>
<td>CVE-2016-5244</td>
<td>1 byte</td>
<td>5</td>
<td>Failed</td>
</tr>
</tbody>
</table>
Conclusion

• Proposed a generic approach to exploit uses of uninitialized stack
  • Can effectively analyze stack-based information-leak vulnerabilities
  • Leaked pointer values -> Bypassing KASLR
  • Can help adjust CVSS scores
Haehyun Cho
haehyun@asu.edu
https://haehyun.github.io