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

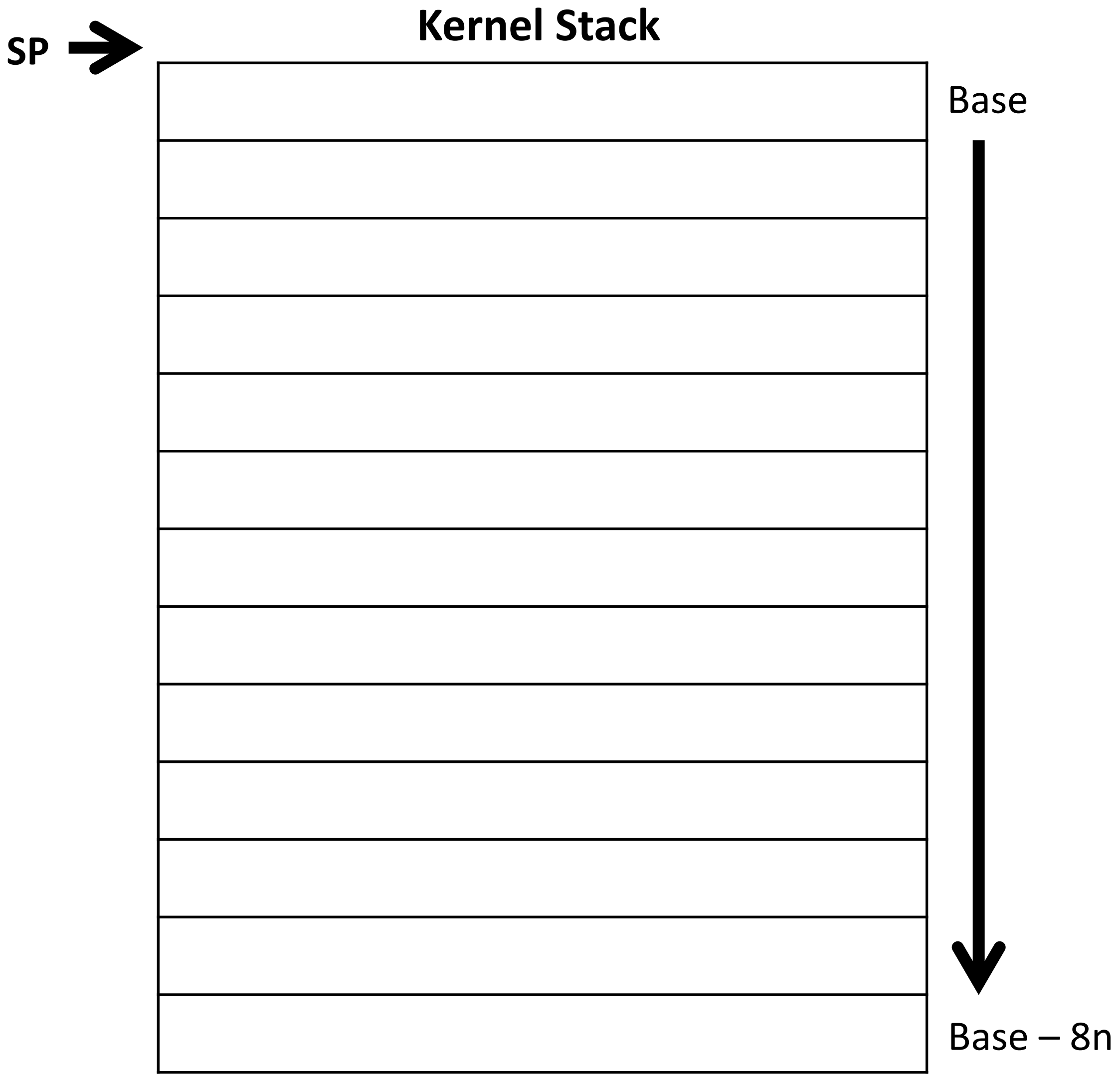
Haehyun Cho, Jinbum Park, Joonwon Kang, Tiffany Bao
Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, Gail-Joon Ahn

ASU CENTER FOR CYBERSECURITY & DIGITAL FORENSICS

SAMSUNG Research

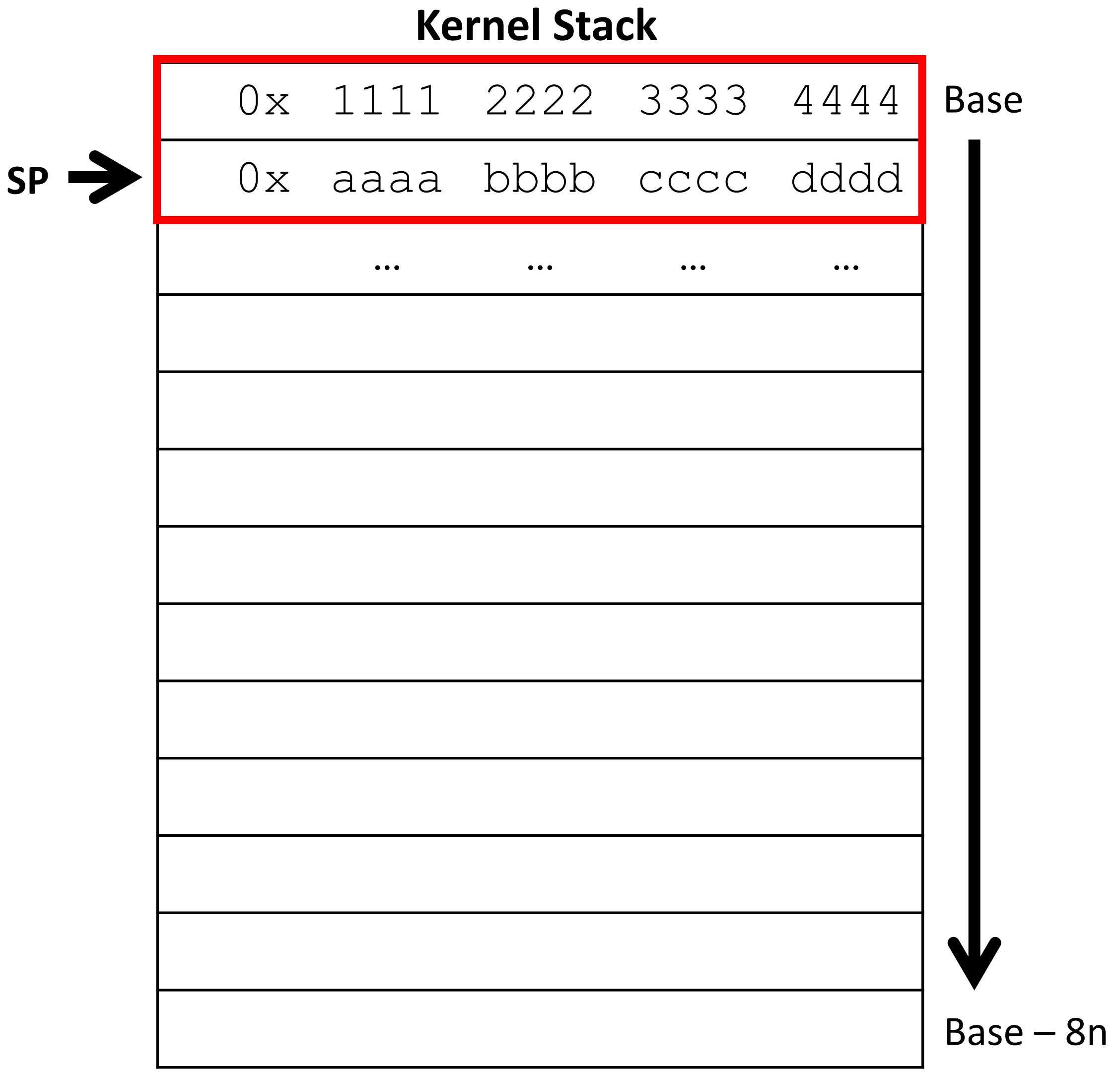
Uninitialized variables in the stack

```
void func () {
    int num;
    int ret;
    …
}
```



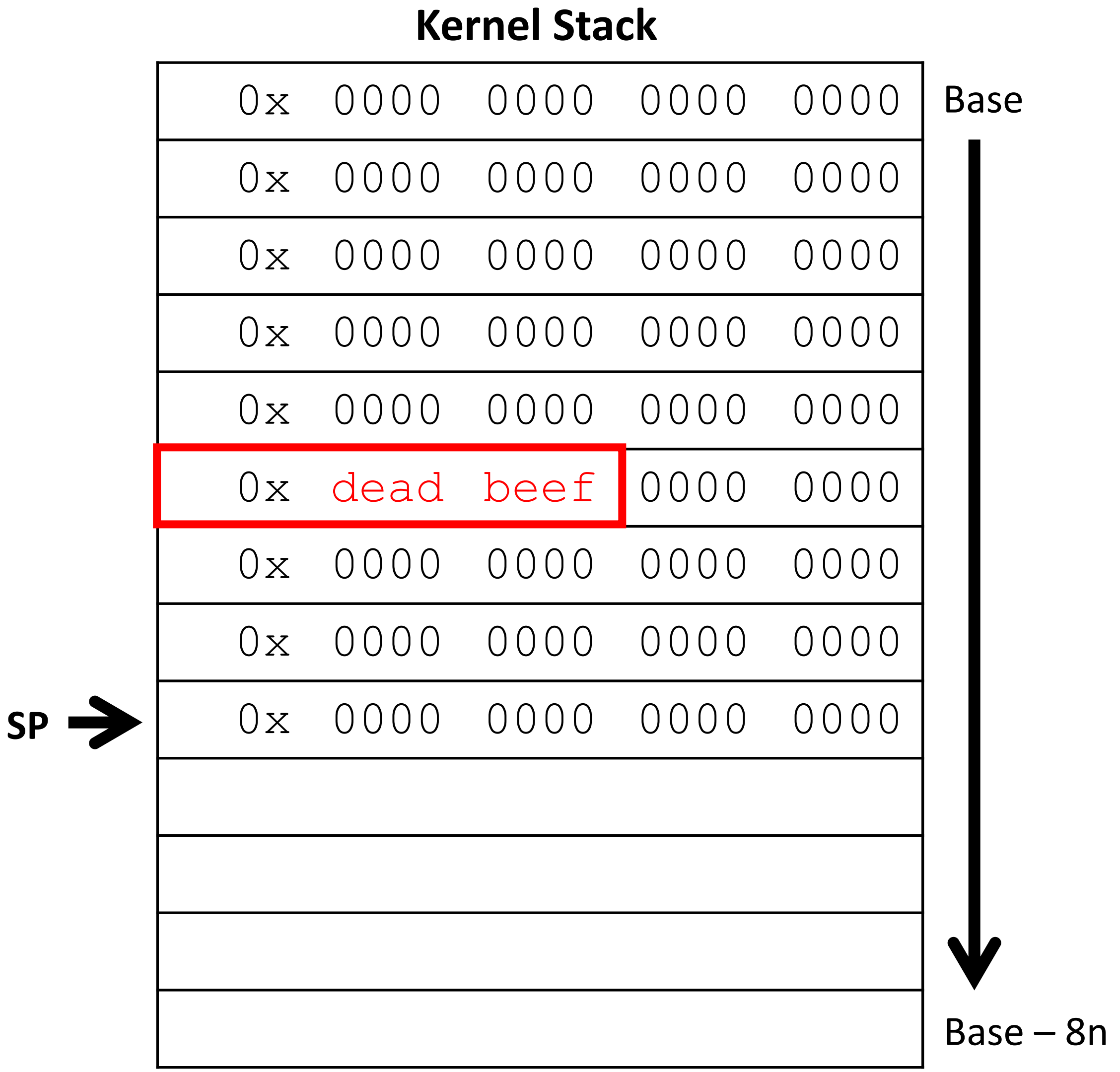
Uninitialized variables in the stack

```
void func () {
    int num;
    int ret;
    …
}
```



Uninitialized variables in the stack

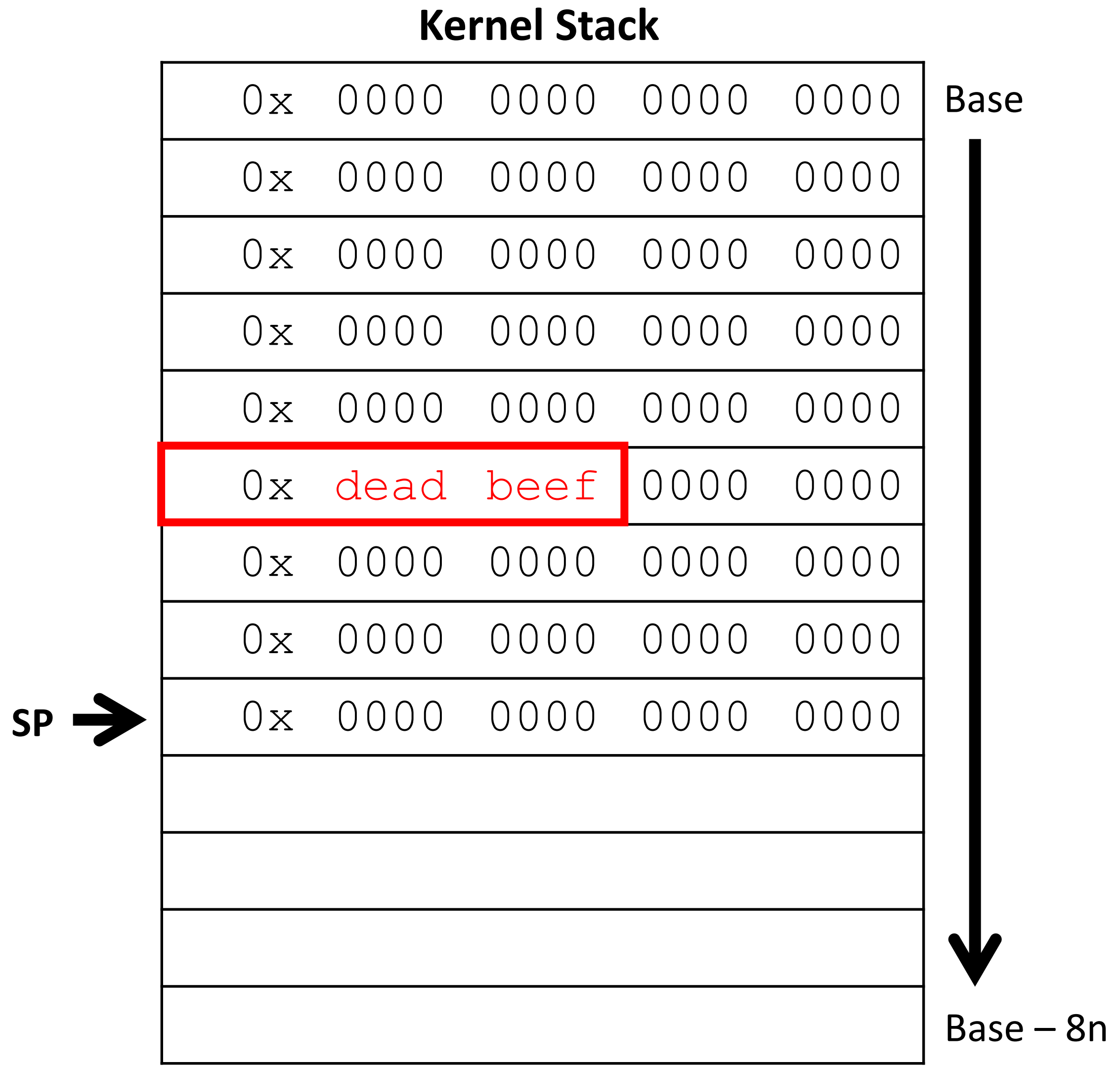
```
void func () {
    int num = 0;
    int ret = 0;
    struct data_struct = {0,};
    …
}
```



Unexpected information leaks

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

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



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)

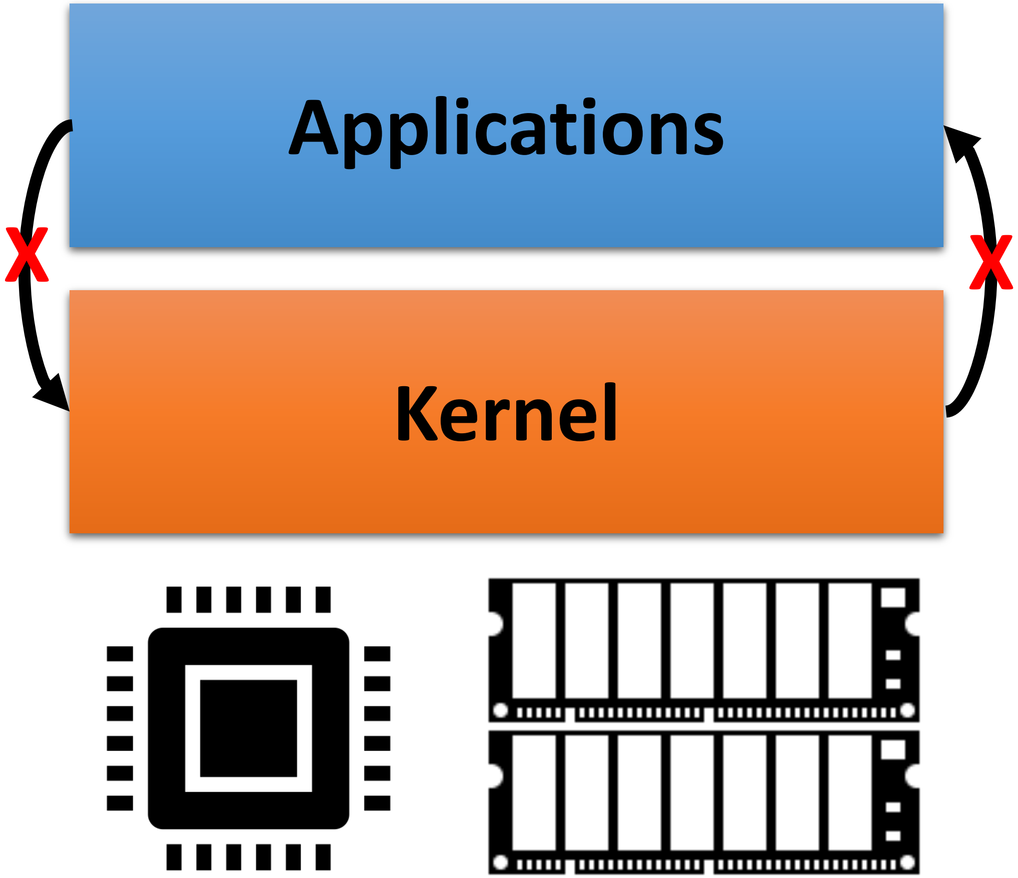
```
/* file: net/core/rtnetlink.c */
static int rtnl_fill_link_ifmap(struct sk_buff *skb, struct net_device *dev)
{
    //all fields in the map object are initialized
    struct rtnl_link_ifmap map = {
        .mem_start    = dev->mem_start,
        .mem_end      = dev->mem_end,
        .base_addr  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

+ 4 padding bytes

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].

[1] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M Frans Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In Proceedings of the 2nd Asia-Pacific Work- shop on Systems (APSys), Shanghai, China, July 2011.
[2] KernelMemorySanitizer, a detector of uses of uninitialized memory in the Linux kernel. https://github.com/google/kmsan.
[3] Kangjie Lu, Marie-Therese Walter, David Pfaff, Ste- fan Nümberger, Wenke Lee, and Michael Backes. Un- leashing Use-Before-Initialization Vulnerabilities in the Linux Kernel Using Targeted Stack Spraying. In Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, February–March 2017.

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

[1] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M Frans Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In Proceedings of the 2nd Asia-Pacific Work- shop on Systems (APSys), Shanghai, China, July 2011.
[2] KernelMemorySanitizer, a detector of uses of uninitialized memory in the Linux kernel. https://github.com/google/kmsan.
[3] Kangjie Lu, Marie-Therese Walter, David Pfaff, Ste- fan Nümberger, Wenke Lee, and Michael Backes. Un- leashing Use-Before-Initialization Vulnerabilities in the Linux Kernel Using Targeted Stack Spraying. In Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, February–March 2017.

Survey result on information leak CVEs

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

	Total	Stack-base	Heap-base	# of exploits
# of CVEs	87	76	11	0

- 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"

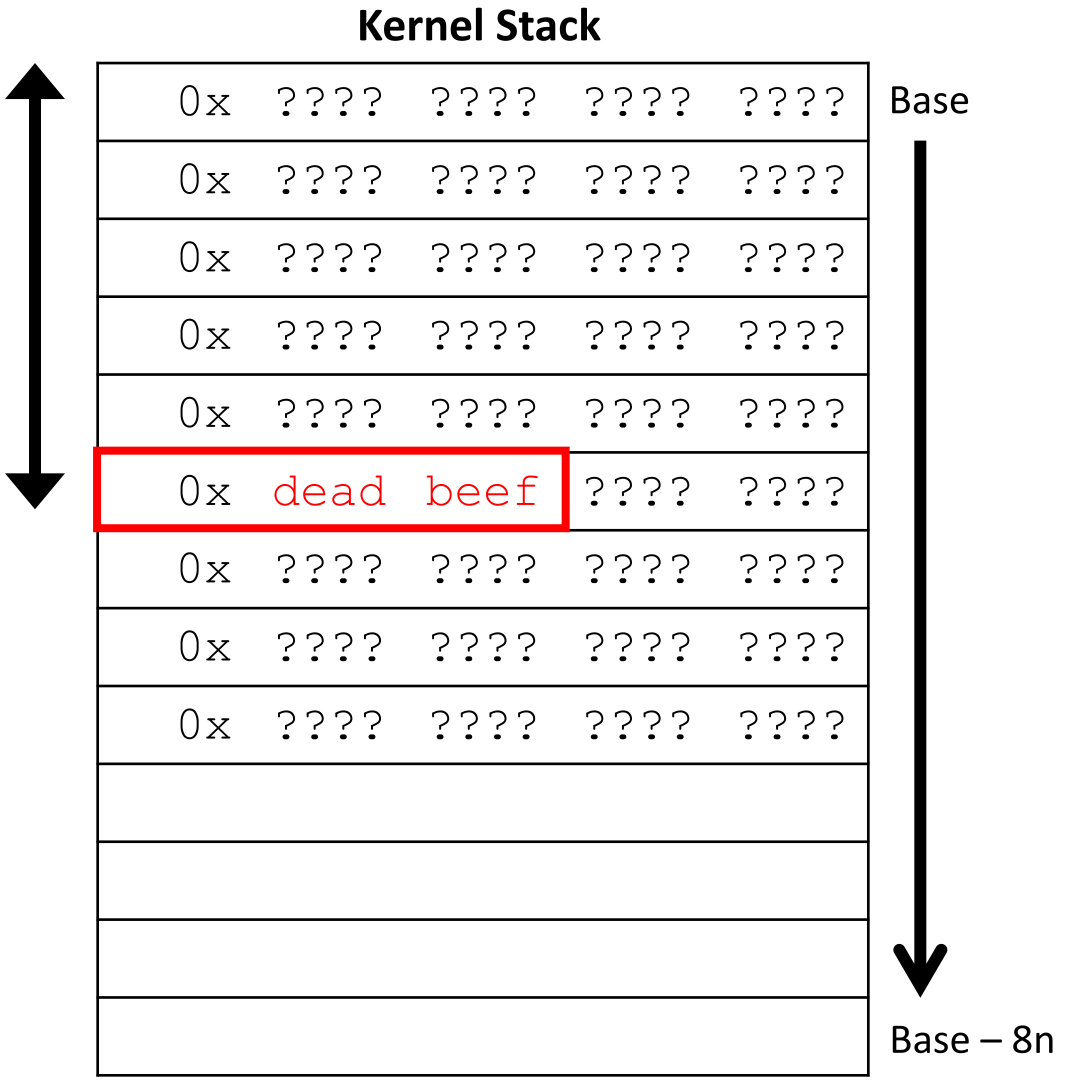
Our Goal

- 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

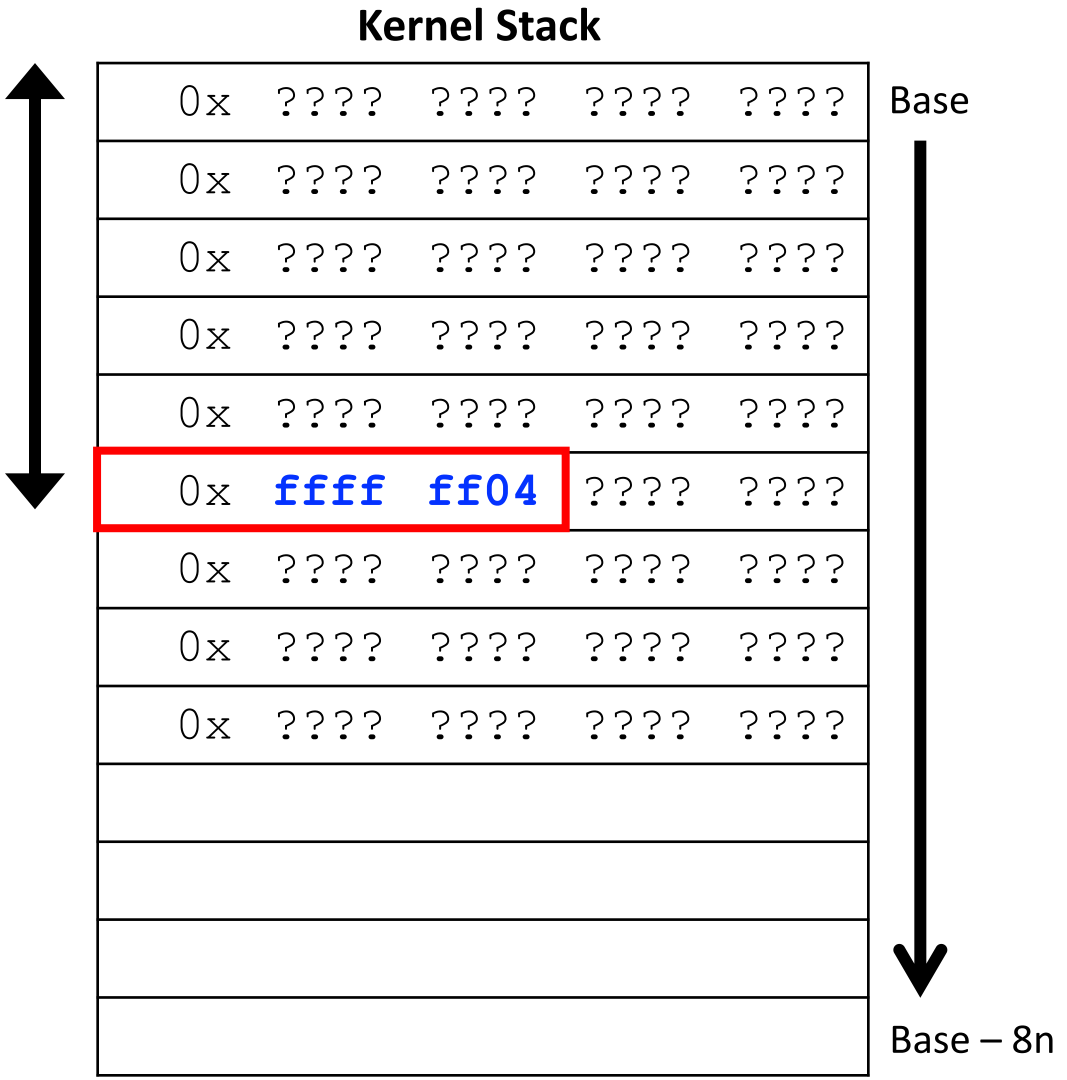
- Computing the offset to uninitialized data from the kernel stack base.



Challenges in Exploitation

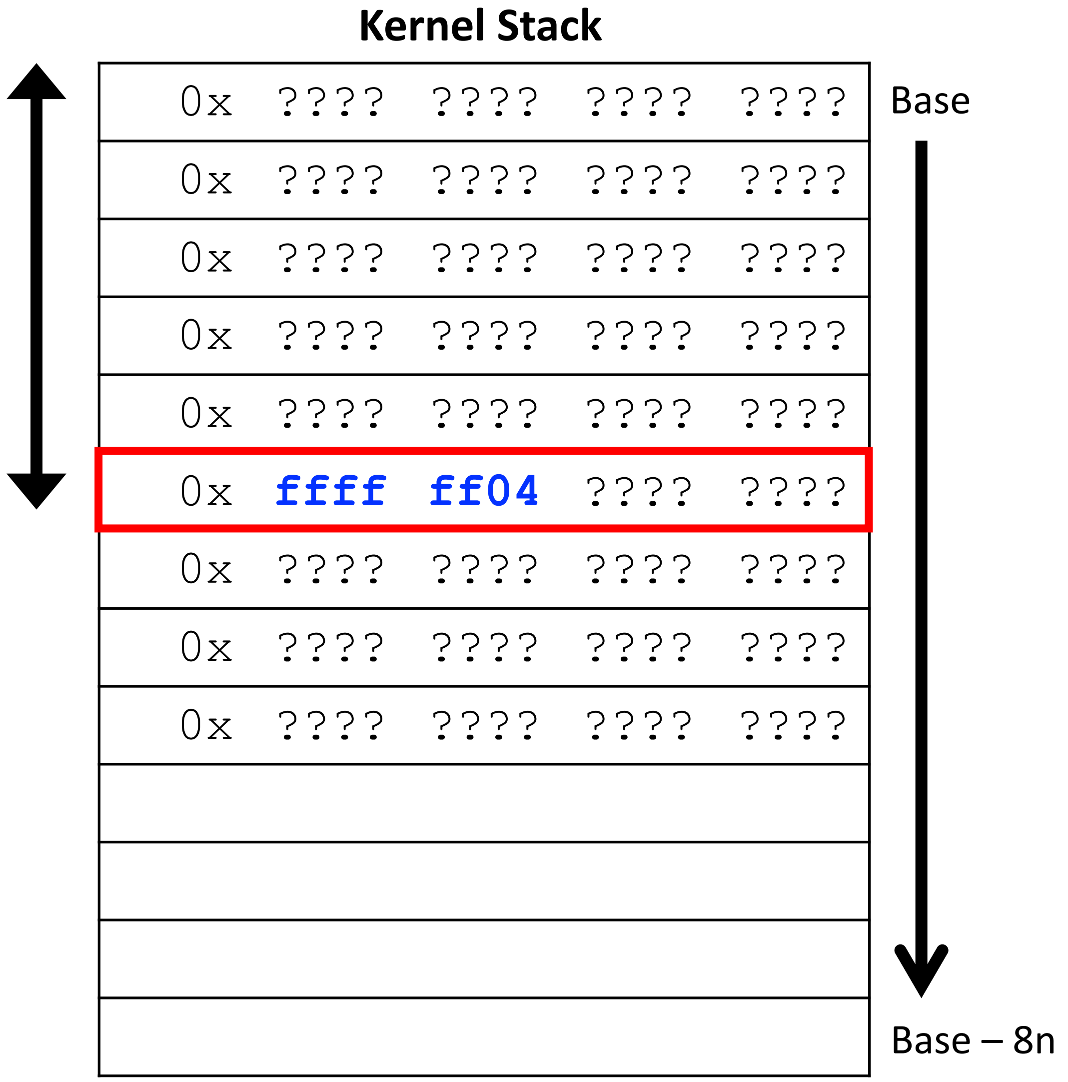
- Computing the offset to uninitialized data from the kernel stack base.

- Storing kernel pointer values at a leak offset.

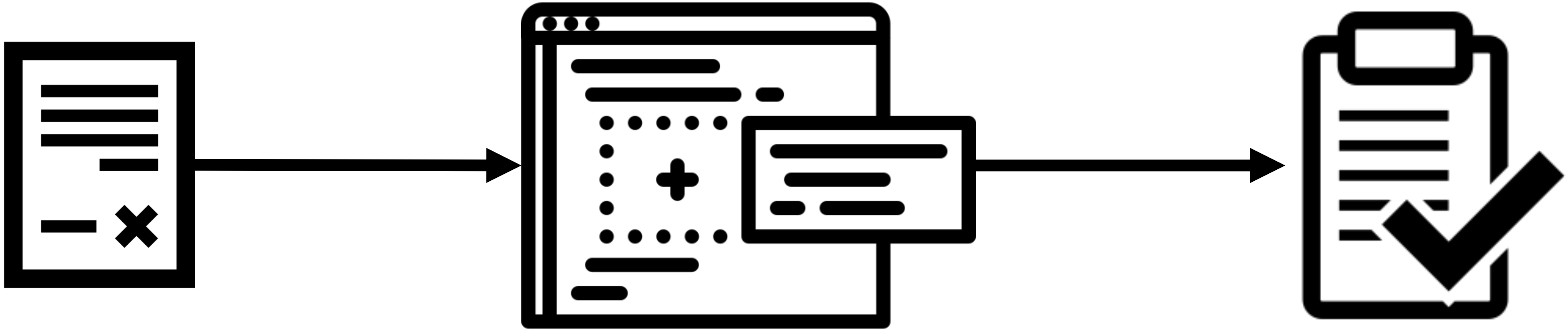


Challenges in Exploitation

- 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

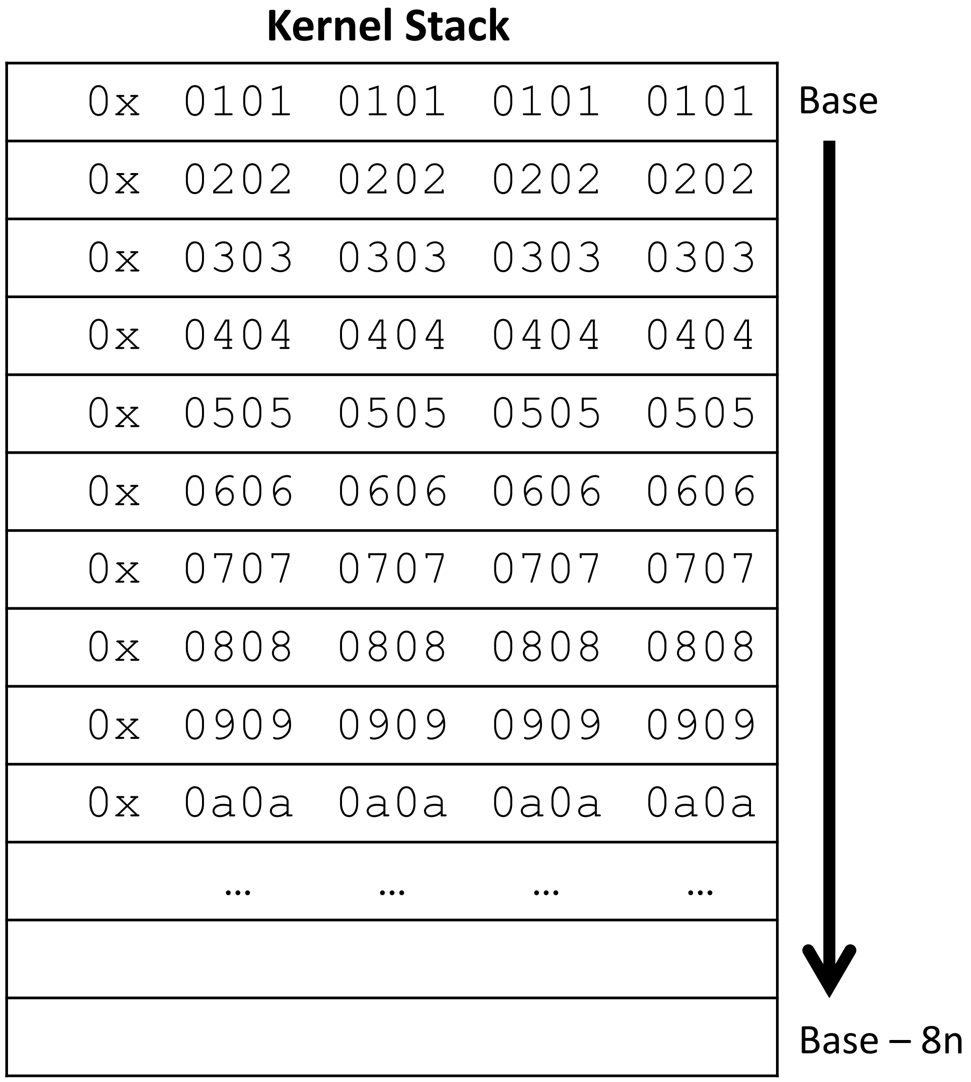


PoVs → **Analysis** → **Exploitability and exploits**

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

`0x 0404 0404 ???? ????`

Kernel Stack

0x	????	????	????	????	Base
0x	????	????	????	????	
0x	????	????	????	????	
0x	**0404**	**0404**	????	????	
0x	????	????	????	????	
0x	????	????	????	????	
0x	????	????	????	????	
0x	????	????	????	????	
0x	????	????	????	????	
0x	????	????	????	????	
	…	…	…	…	
					Base – 8n

Computing the Leak Offset

Stack footprinting

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

0x	**0404**	**0404**	????	????

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

Kernel Stack

0x	????	????	????	????	Base
0x	????	????	????	????	
0x	????	????	????	????	
0x	**0404**	**0404**	????	????	
0x	????	????	????	????	
0x	????	????	????	????	
0x	????	????	????	????	
0x	????	????	????	????	
0x	????	????	????	????	
0x	????	????	????	????	
	…	…	…	…	
					Base – 8n

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

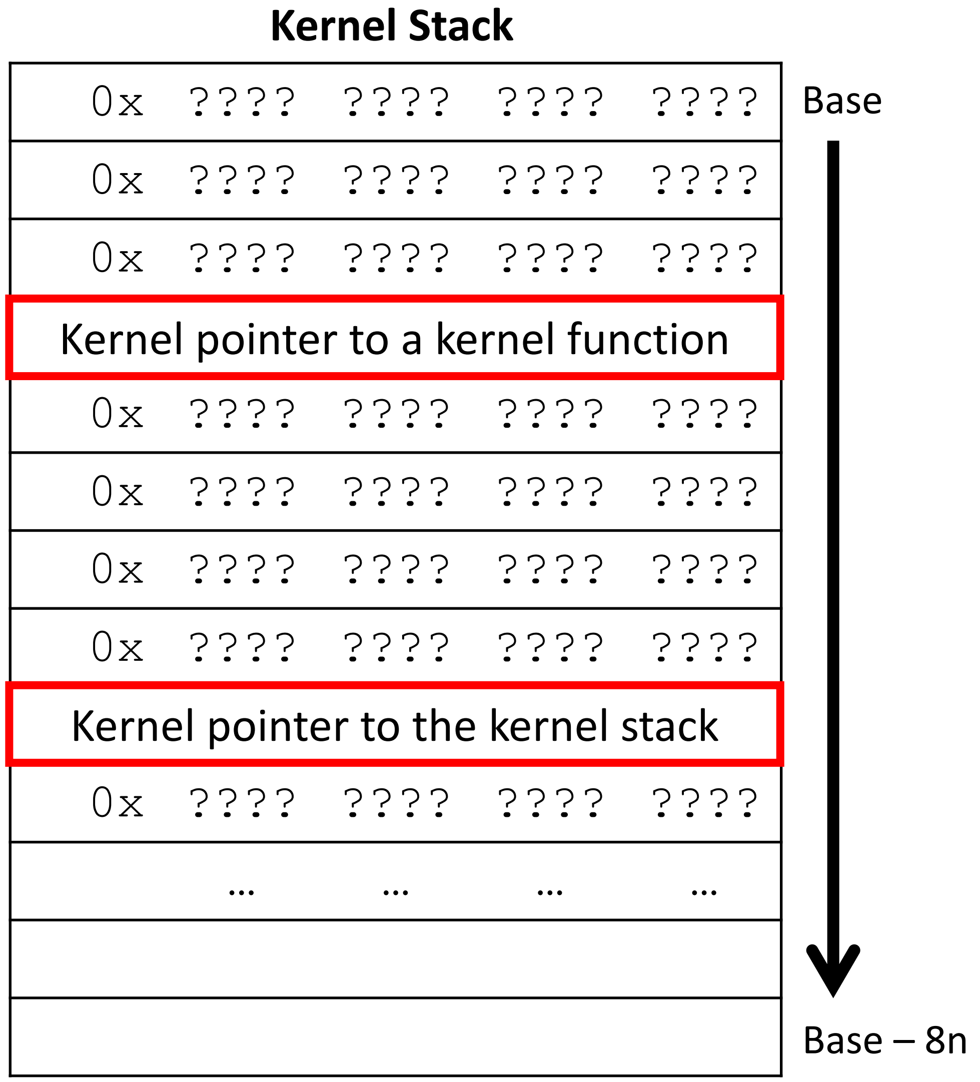
1. Fill the kernel stack

Kernel Stack

0x 1122 3344 5566 7788	Base
0x 1122 3344 5566 7788	
0x 1122 3344 5566 7788	
0x 1122 3344 5566 7788	
0x 1122 3344 5566 7788	
0x 1122 3344 5566 7788	
0x 1122 3344 5566 7788	
0x 1122 3344 5566 7788	
0x 1122 3344 5566 7788	
0x 1122 3344 5566 7788	
… … … …	
	Base – 8n

Syscall Testing with the LTP

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



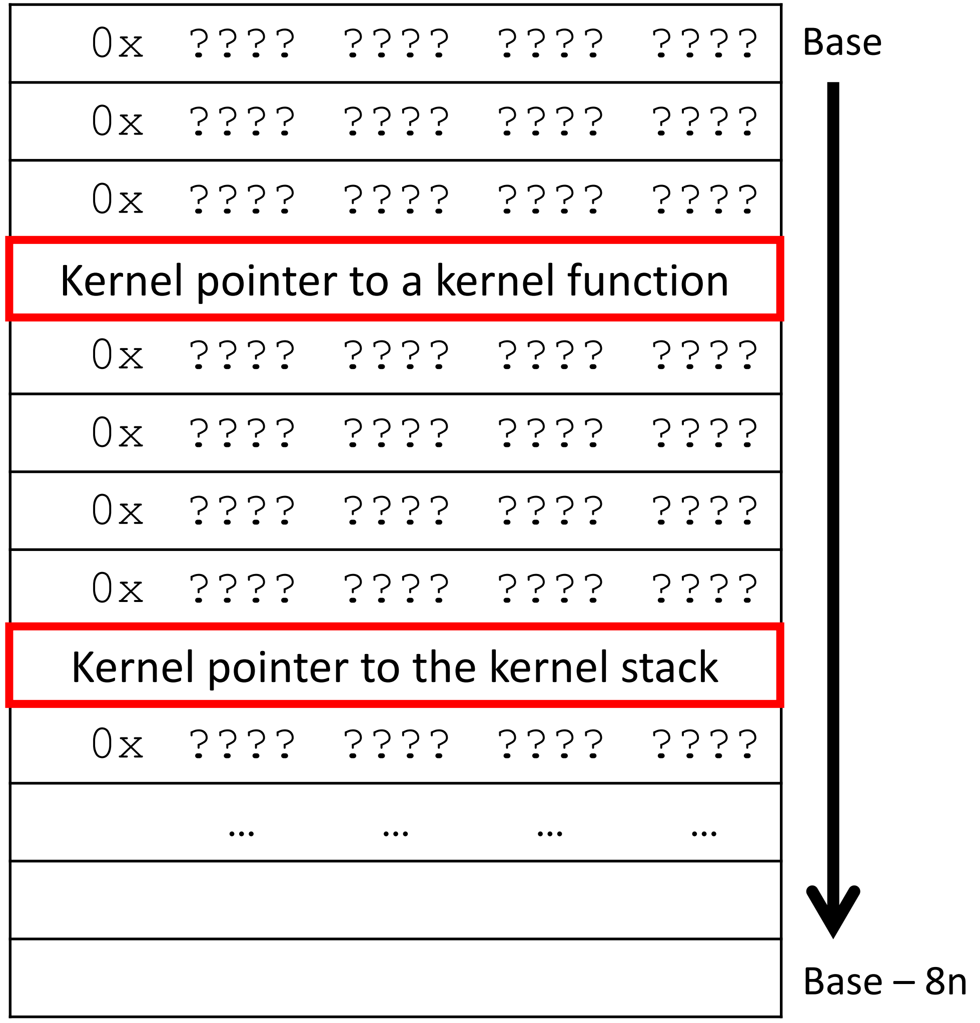
Syscall Testing with the LTP

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

```
Offset  : Base – 24
Type    : Kernel code
Syscall : mmap
Args    : 0,8,0,0,-1,0
```

```
Offset  : Base – 64
Type    : Kernel stack
Syscall : mmap
Args    : 0,8,0,0,-1,0
```

Kernel Stack

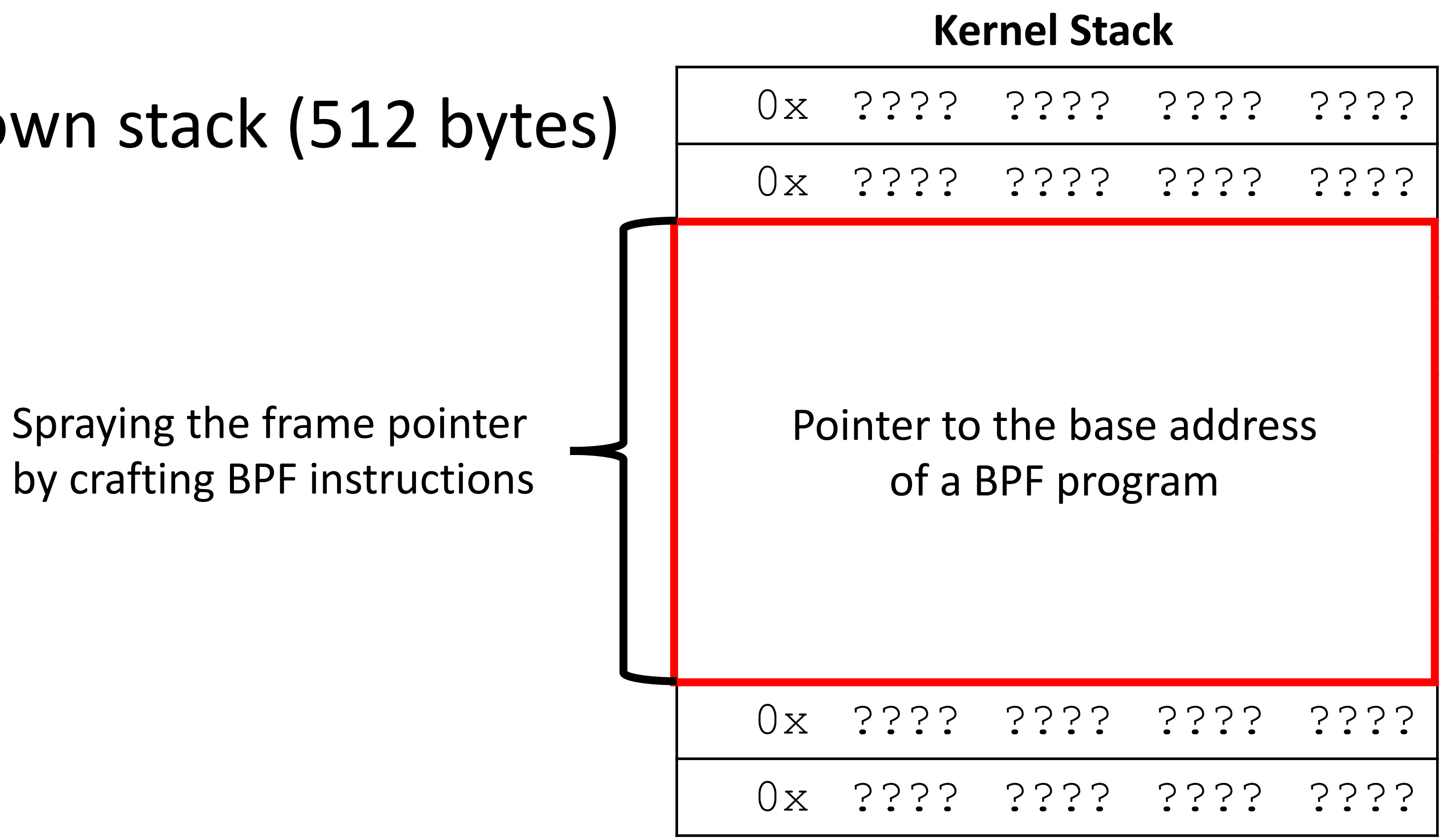


Stack Spraying via BPF

- 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)

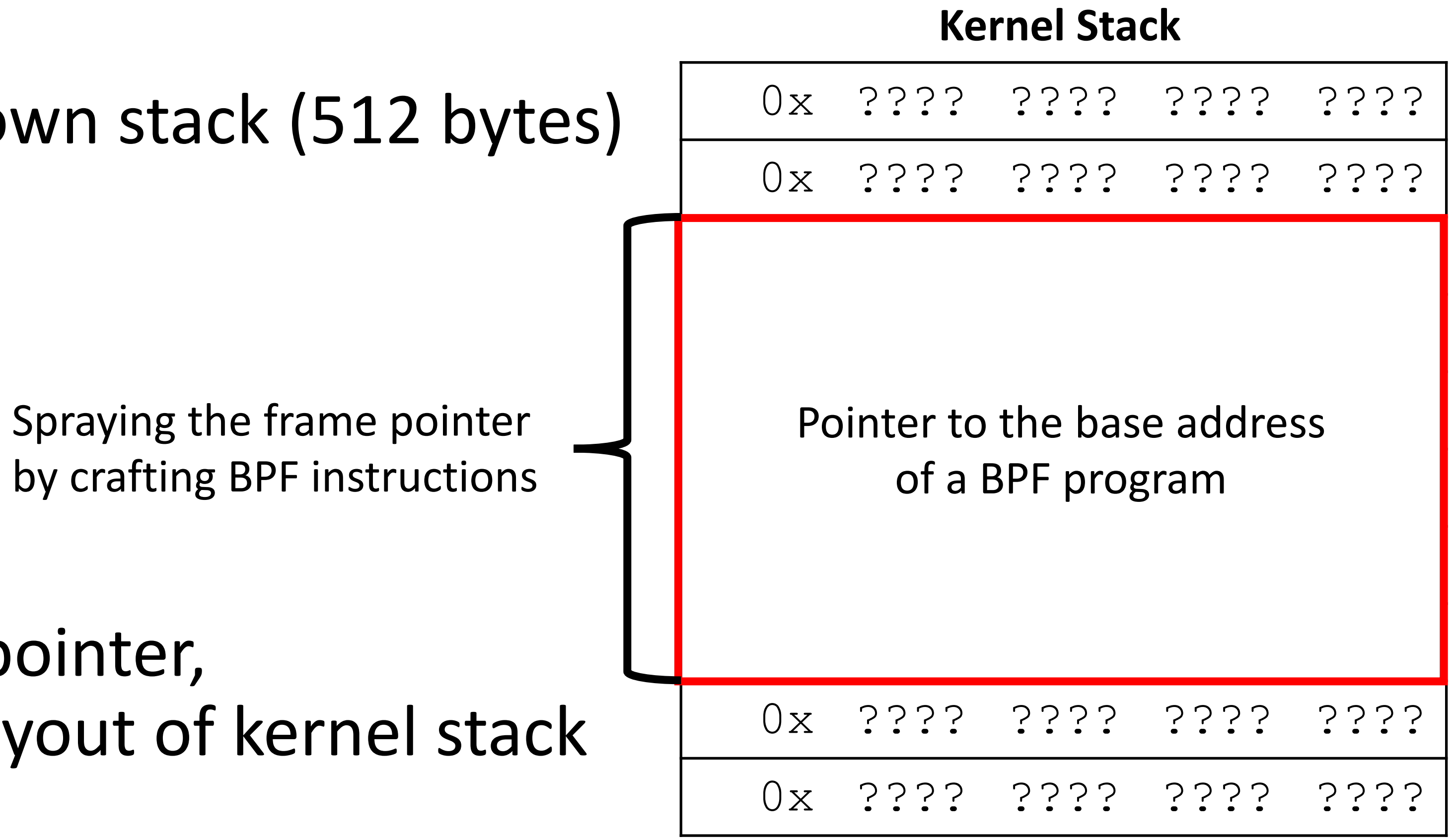


Stack Spraying via BPF

- 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 ff04 2000 0000`

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

Hidden data < 0x 3000 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!

`0x ffff ff04 2000 0000`

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

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

  Hidden data < 0x 3000 0000

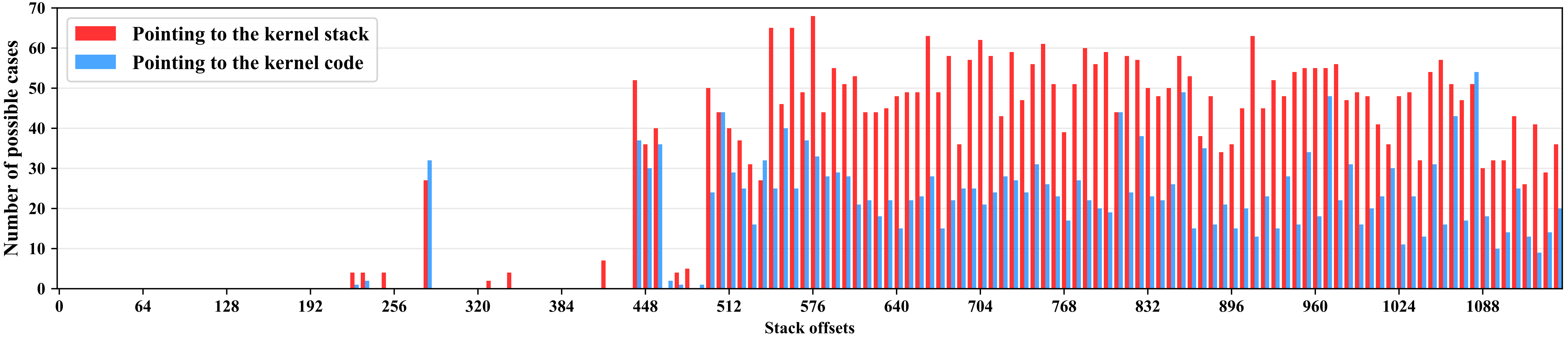
  spraying (FP — 0x0000 0000 1234 0000)

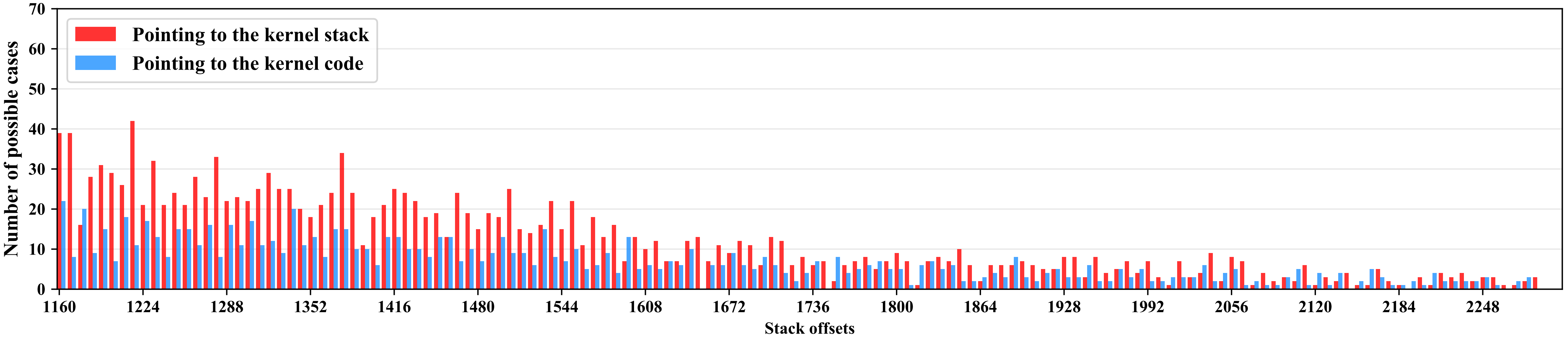
  `0x ffff ff04 ???? ????`

  Hidden data > 0x 1234 0000

Evaluation

Finding pointers with the LTP framework





Summary of exploitation results

Vulnerability	Leak Size	CVSS	Exploitation Result
CVE-2018-11580	4 bytes	2.1	Bypass KASLR
CVE-2016-4569	4 bytes	2.1	Bypass KASLR
Fixes: 372f525	4 bytes	N/A	Bypass KASLR
CVE-2016-4486	4 bytes	2.1	Reveal the kernel stack base
CVE-2016-5244	1 byte	5	Failed

Summary of exploitation results

Vulnerability	Leak Size	CVSS	Exploitation Result
CVE-2018-11580	4 bytes	2.1	Bypass KASLR
CVE-2016-4569	4 bytes	2.1	Bypass KASLR
Fixes: 372f525	4 bytes	N/A	Bypass KASLR
CVE-2016-4486	4 bytes	2.1	Reveal the kernel stack base
CVE-2016-5244	1 byte	5	Failed

Summary of exploitation results

Vulnerability	Leak Size	CVSS	Exploitation Result
CVE-2018-11580	4 bytes	2.1	Bypass KASLR
CVE-2016-4569	4 bytes	2.1	Bypass KASLR
Fixes: 372f525	4 bytes	N/A	Bypass KASLR
CVE-2016-4486	4 bytes	2.1	Reveal the kernel stack base
CVE-2016-5244	1 byte	5	Failed

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