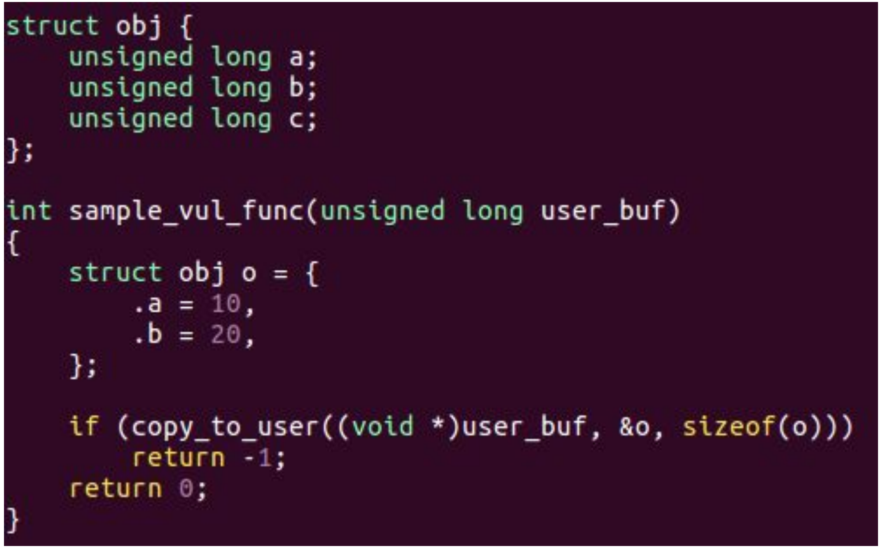
Leak kernel pointer by exploiting uninitialized uses in Linux kernel

Jinbum Park
jinb-park.github.io
jinb.park7@gmail.com

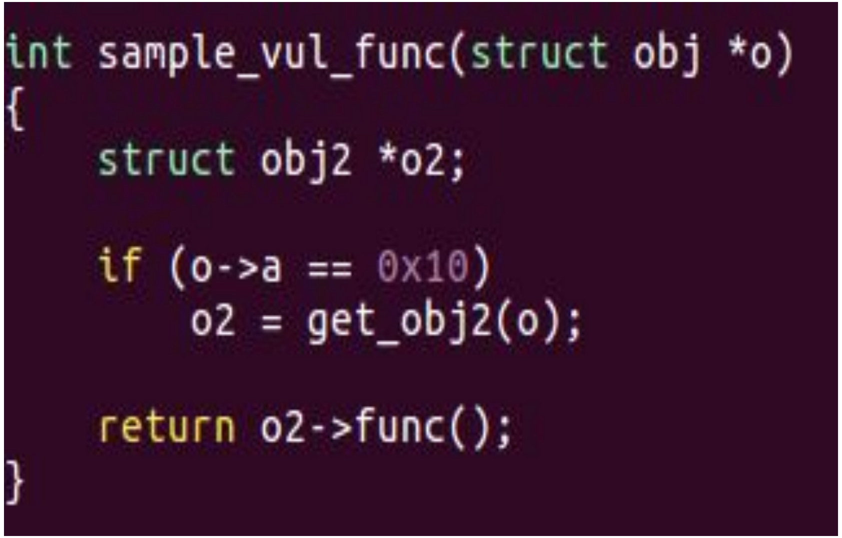
Background

Uninitialized use vulnerability



```
struct obj {
    unsigned long a;
    unsigned long b;
    unsigned long c;
};

int sample_vul_func(unsigned long user_buf)
{
    struct obj o = {
        .a = 10,
        .b = 20,
    };

    if (copy_to_user((void *)user_buf, &o, sizeof(o)))
        return -1;
    return 0;
}
```

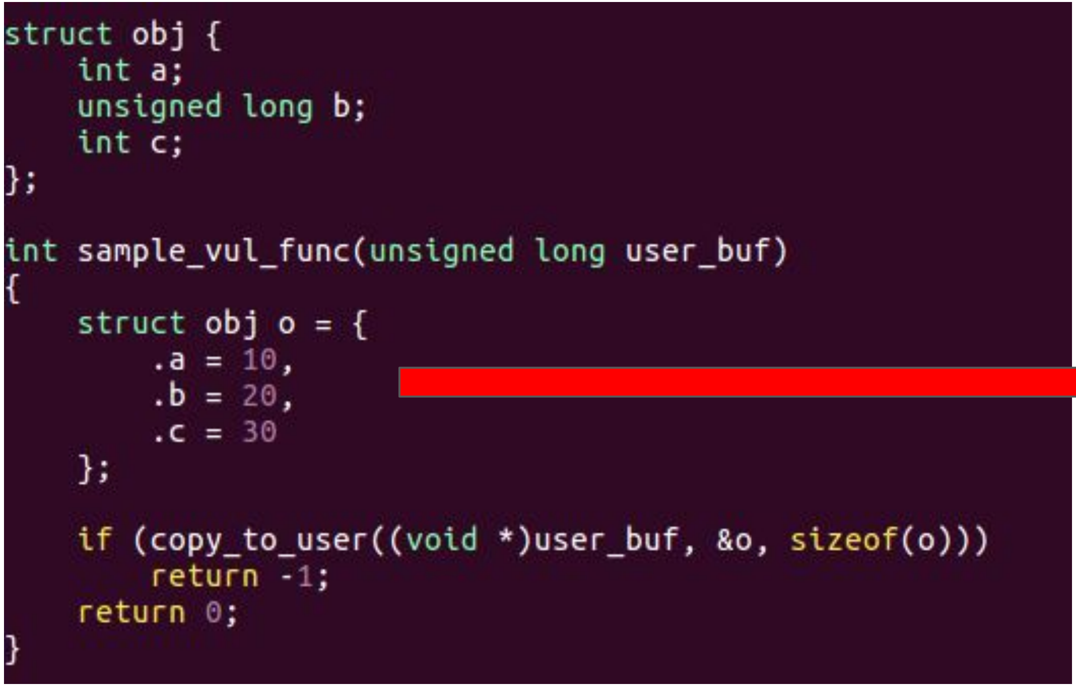
Uninitialized use => Information leak



```
int sample_vul_func(struct obj *o)
{
    struct obj2 *o2;

    if (o->a == 0x10)
        o2 = get_obj2(o);

    return o2->func();
}
```

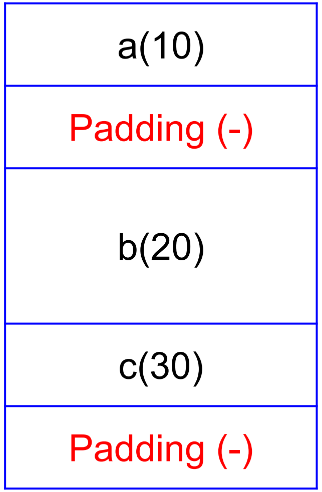
Uninitialized use => Pointer dereference

Uninitialized use vulnerability



```
struct obj {
    int a;
    unsigned long b;
    int c;
};

int sample_vul_func(unsigned long user_buf)
{
    struct obj o = {
        .a = 10,
        .b = 20,
        .c = 30
    };

    if (copy_to_user((void *)user_buf, &o, sizeof(o)))
        return -1;
    return 0;
}
```

No problem?



a(10)

Padding (-)

b(20)

c(30)

Padding (-)

sizeof(o) == 24 byte!!

leak due to padding!!

Prior works

No comprehensive research on exploitations of uninitialized use for Information Leak!!

1. **UniSan (ACM CCS 2016)**
   - Bug finding : O
   - Exploitation : X
2. **Unleashing Use-Before-Initialization Vulnerabilities in the Linux kernel Using Targeted Stack Spraying (NDSS 2017)**
   - Bug finding : X
   - Exploitation : Pointer dereference from stack
   - Note : Not dealing with Information Leak
3. **Exploitations of Uninitialized Uses on macOS Sierra (USENIX WOOT 2017)**
   - Bug finding : X
   - Exploitation : Information Leak from heap, Pointer dereference from stack
   - Note : Dependent on specific vulnerability. Not dealing with Information Leak from stack.

Uninitialized use CVEs (Information Leak)

- We investigated Uninitialized use CVEs in Linux kernel reported from 2015 to 2018.
- Investigated 22 CVEs manually.
- A lot of vulnerabilities have been fixed upstream, but not assigned as CVEs.

- **Where does it occurs from which memory type?**

  -- **Stack : 17 CVEs (77.3%)** / Heap : 5 CVEs (22.7%)

- **Leak size (CVEs from Stack)**

  -- **Less than 8byte : 10 CVEs (58.8%)** / Greater than 8byte : 7 CVEs (41.2%)

- **Leak size (CVEs from Heap)**

  -- Less than 8byte : 0 CVEs (0%) / **Greater than 8byte : 5 CVEs (100%)**

Type of kernel pointer we’re interested in

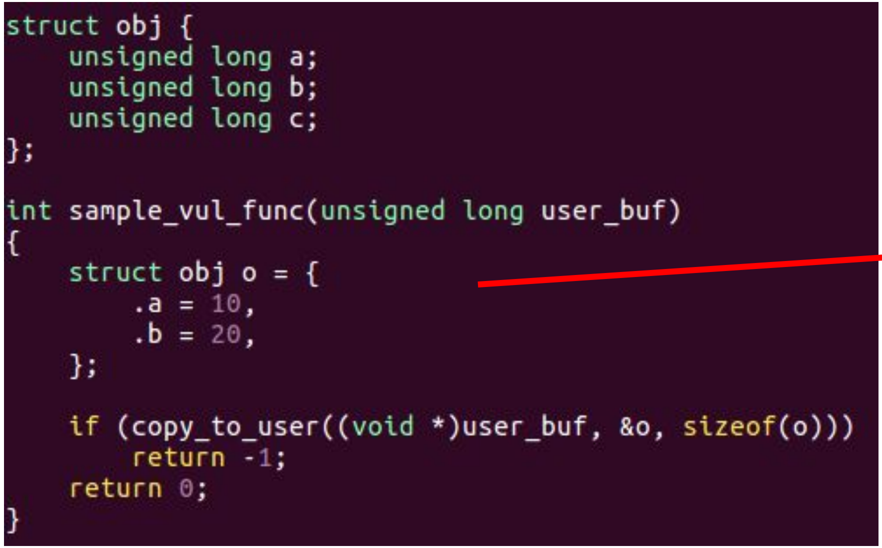
- We define sensitive kernel pointer as follows.
  - Pointer to kernel code. (bypass KASLR)
  - Pointer to kernel stack. (contains a lot of sensitive data, thread_info)
  - Pointer to kernel object.

Goal

- Defines common exploitation steps which is not dependent on specific vulnerability.
- Defines common challenges for successful exploitation.
- Presents generic methods and tools for solving the challenges.
- Exploits real-world vulnerabilities with the methods and tools.

Exploitation steps

Sample Vul - Uninitialized use from stack



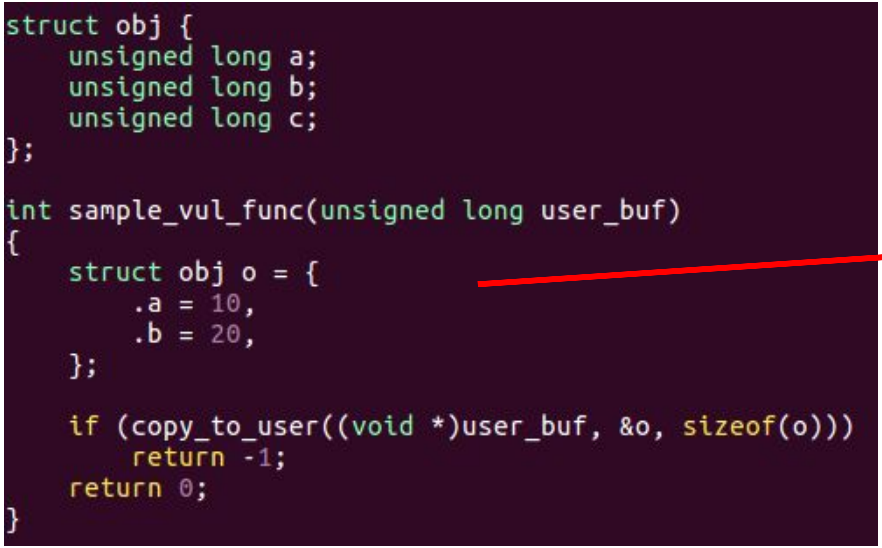
```
struct obj {
    unsigned long a;
    unsigned long b;
    unsigned long c;
};

int sample_vul_func(unsigned long user_buf)
{
    struct obj o = {
        .a = 10,
        .b = 20,
    };

    if (copy_to_user((void *)user_buf, &o, sizeof(o)))
        return -1;
    return 0;
}
```

Kernel stack

a	10
b	20
c	??? random kernel data

- Copy “obj” including not initialized o.c to user space.
- Then, 8 bytes arbitrary kernel stack memory leak happens.
- Problem?? Since the leaked data is random data, Attacker can’t utilize the data.

What should we do



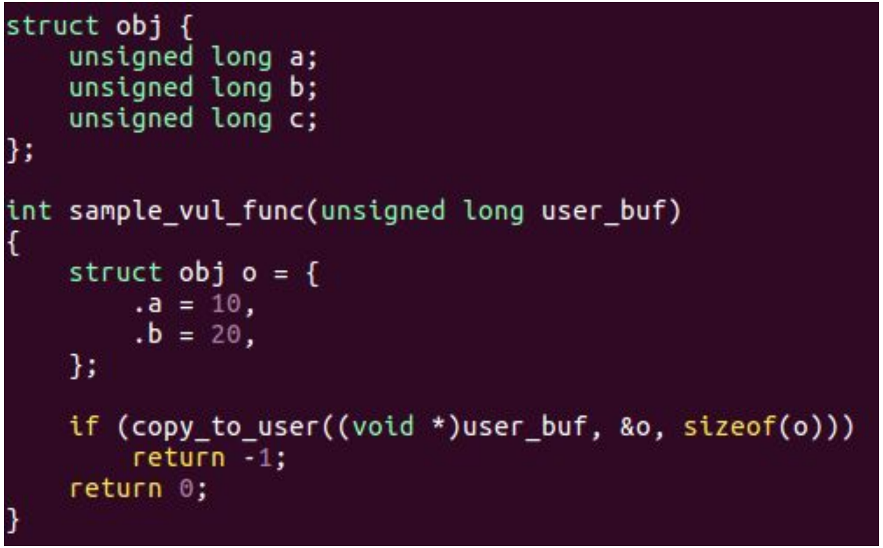
```
struct obj {
    unsigned long a;
    unsigned long b;
    unsigned long c;
};

int sample_vul_func(unsigned long user_buf)
{
    struct obj o = {
        .a = 10,
        .b = 20,
    };

    if (copy_to_user((void *)user_buf, &o, sizeof(o)))
        return -1;
    return 0;
}
```

Kernel stack

a	10
b	20
c	**Sensitive kernel pointer**

- Put “Sensitive kernel pointer” on the memory “o.c” prior to trigger vulnerability.

Exploitation steps



```c
struct obj {
    unsigned long a;
    unsigned long b;
    unsigned long c;
};

int sample_vul_func(unsigned long user_buf)
{
    struct obj o = {
        .a = 10,
        .b = 20,
    };

    if (copy_to_user((void *)user_buf, &o, sizeof(o)))
        return -1;
    return 0;
}
```

1. Calculate offset of leaked memory “o.c” from base address. => Leak offset
2. Put sensitive kernel pointer on the Leak offset. (Attacker already knows the type of the kernel pointer.)
3. Trigger vulnerability.
4. See the kernel pointer.

Challenges

Challenges

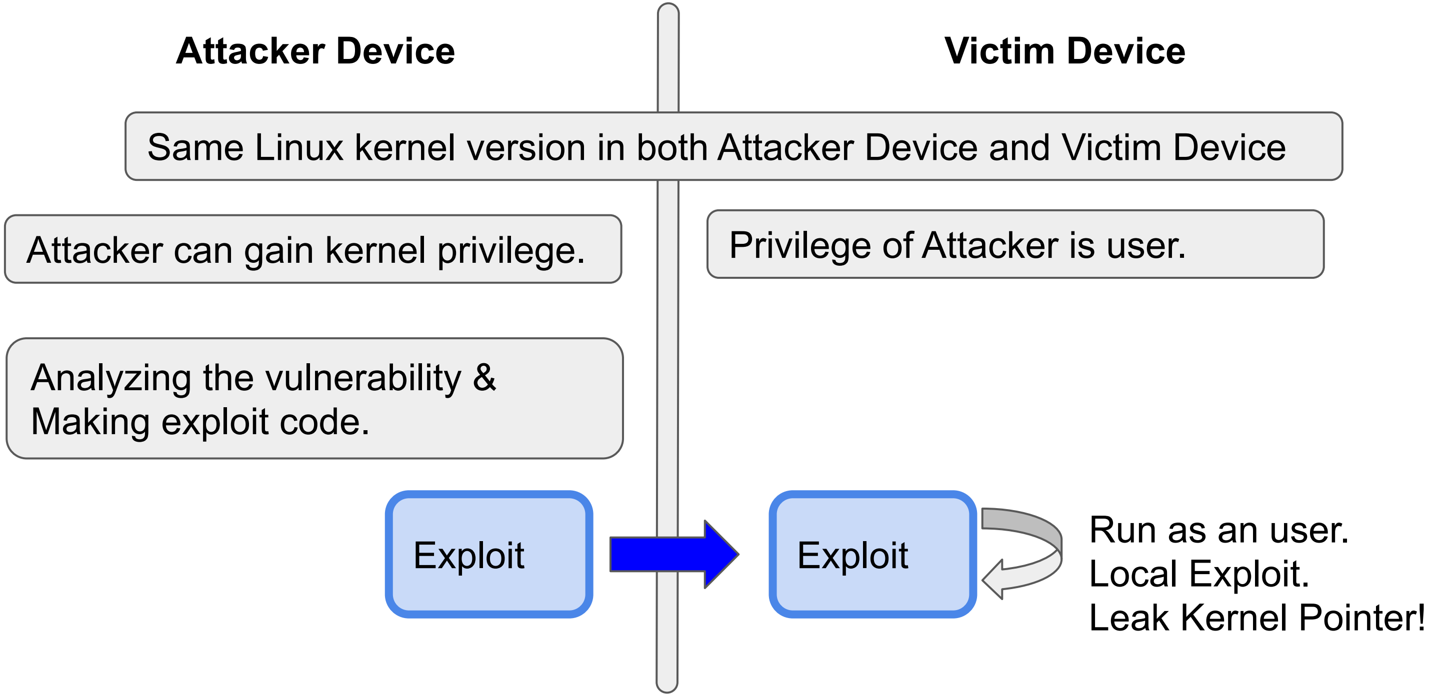
1. Calculate offset of leaked memory “o.c” from base address.
2. Put sensitive kernel pointer on the Leak offset. (Attacker already knows the type of the kernel pointer.)
3. Trigger vulnerability.
4. See the kernel pointer.

1. How to calculate the leak offset? (C-1)
2. How to put sensitive kernel pointer on the leak offset? (C-2)
3. What If leak size is less than 8 bytes? (C-3)

** When sensitive kernel pointer has been overwritten unintentionally. (Failed to solve)

Exploitation Techniques

Assumption & Environment (Confirmed on Ubuntu)



Attacker Device

Victim Device

Same Linux kernel version in both Attacker Device and Victim Device

Attacker can gain kernel privilege.

Privilege of Attacker is user.

Analyzing the vulnerability &
Making exploit code.

Exploit

Exploit

Run as an user.
Local Exploit.
Leak Kernel Pointer!

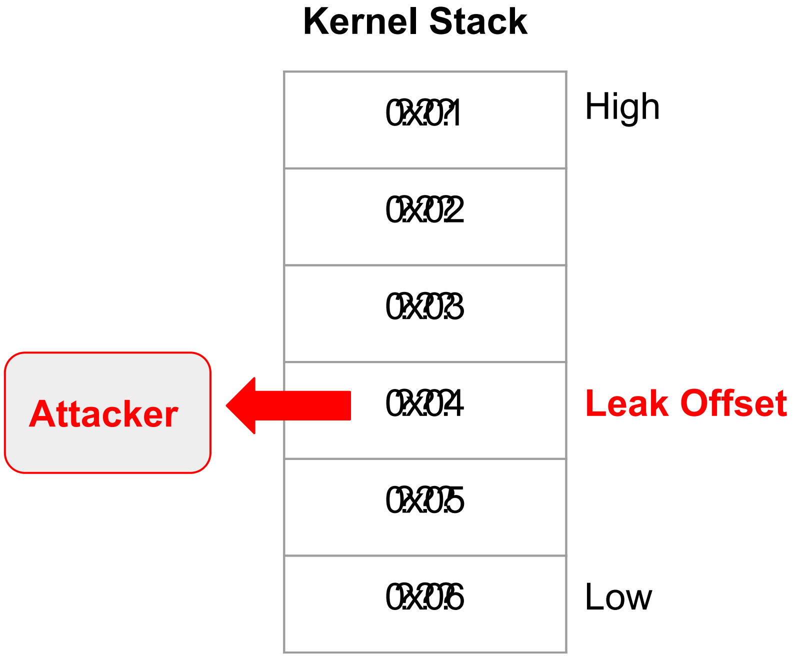
C-1. How to caclulate leak offset?

Footprinting kernel stack - Concept

C-1. How to calculate the leak offset? ---> C-2 ---> C-3

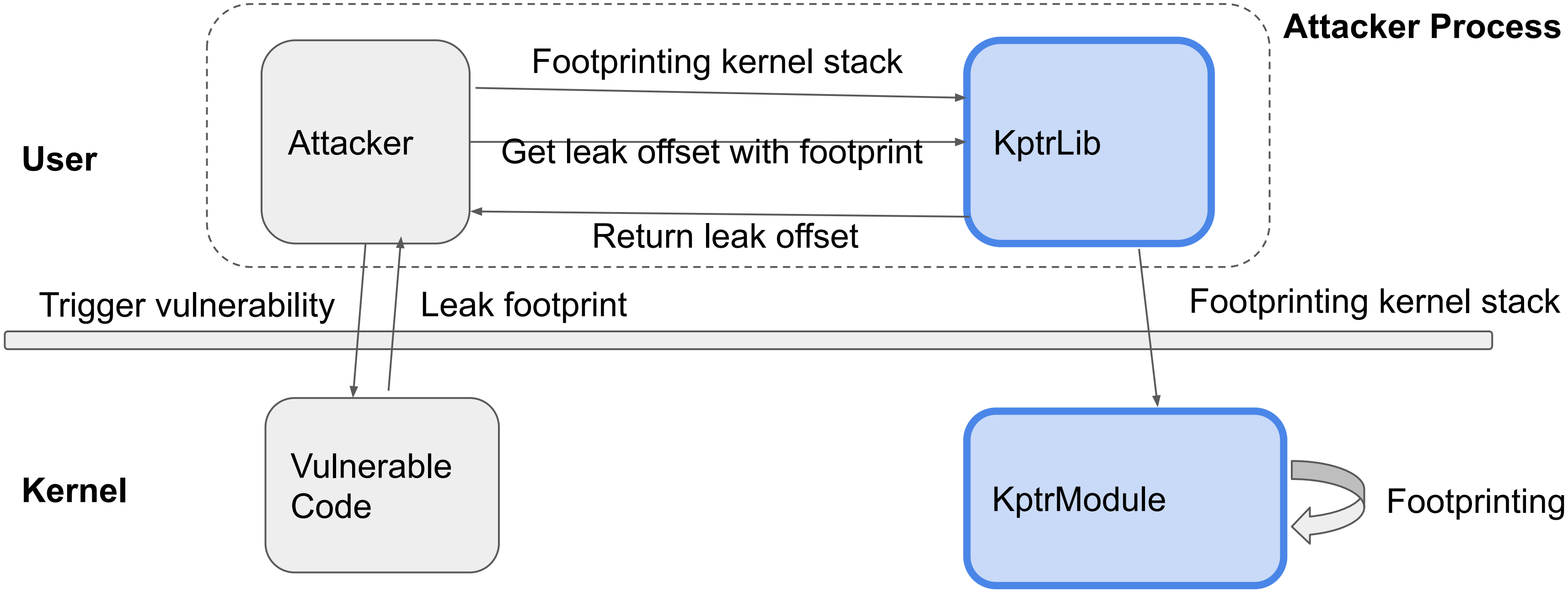
1. Footprinting kernel stack with distance from base
2. Trigger the vulnerability
3. See the footprint.

   Calculate the leak offset based on the footprint.



Footprinting kernel stack - Implementation

C-1. How to calculate the leak offset? ---> C-2 ---> C-3



C-2. How to put sensitive kernel pointer on the Leak Offset?

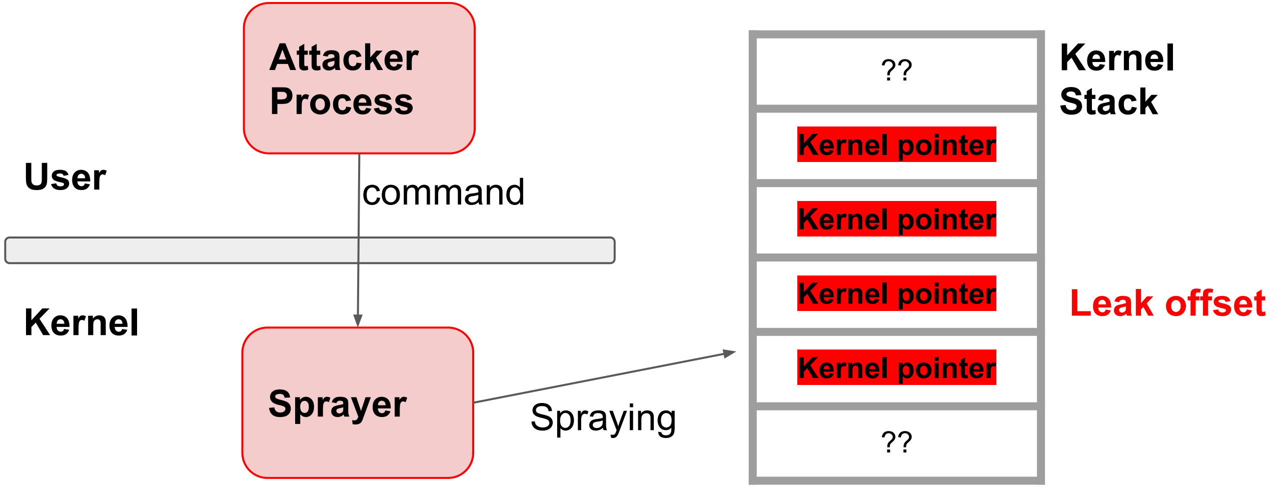
Approach-1. Kernel pointer spraying (KptrSpray)

C-1 ---> **C-2. How to put sensitive kernel pointer on the leak offset?** ---> C-3

- Fill large amount of kernel stack memory with sensitive kernel pointer.
- In hacking community, Spraying generally means that fill memory with the value that attacker knows. (e.g. fake object address, code address)
- But for leaking kernel pointer, Attacker should fill stack memory with the sensitive kernel pointer that attacker doesn’t know.
- So this kind of spraying is a special case. We call it Kernel pointer spraying. (KptrSpray)

Approach-1. Kernel pointer spraying (KptrSpray)

C-1 ---> **C-2. How to put sensitive kernel pointer on the leak offset?** ---> C-3



- Even though an attacker doesn’t know sensitive kernel pointer value, Attacker can spray kernel stack by exploiting “Sprayer” which is one of kernel subsystem.
- We found the “Sprayer” by manual kernel code analysis. The “Sprayer” we found is eBPF.

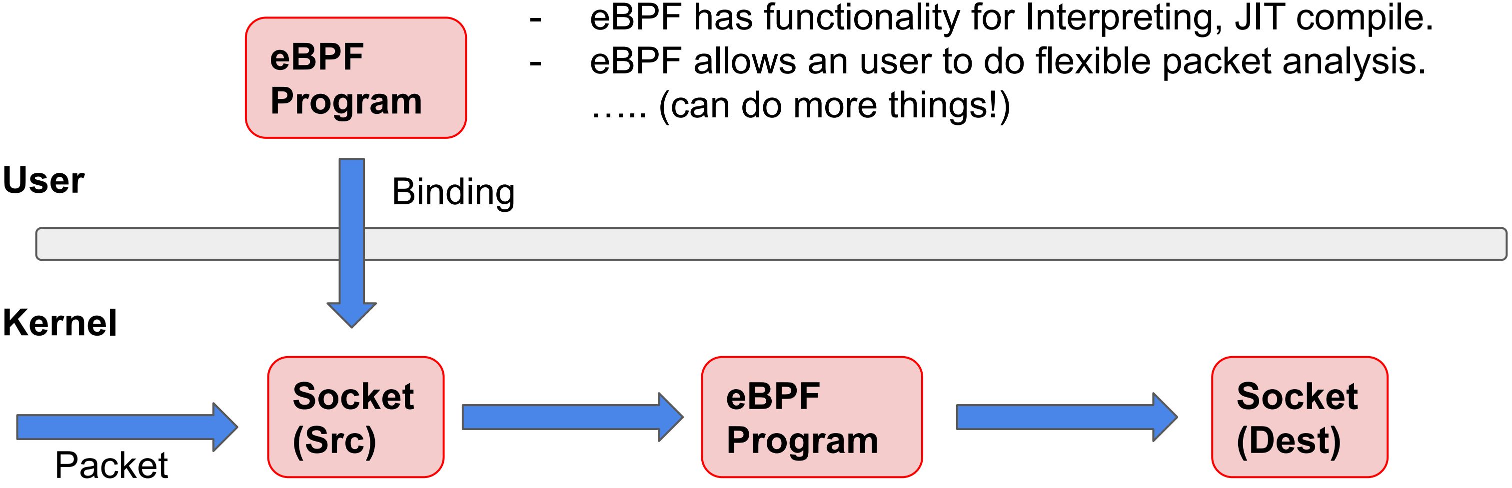
Approach-1. Kernel pointer spraying (KptrSpray)

C-1 ---> **C-2. How to put sensitive kernel pointer on the leak offset?** ---> C-3

What is eBPF?

- eBPF has their own ISA. (Instruction Set Architecture)
- eBPF has functionality for Interpreting, JIT compile.
- eBPF allows an user to do flexible packet analysis.

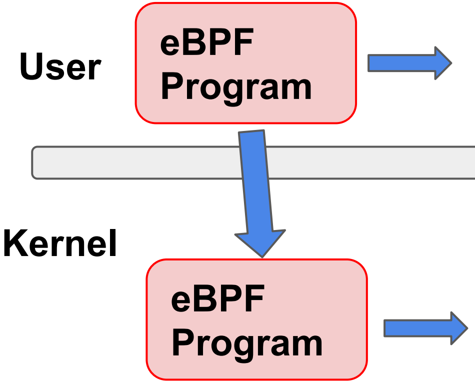
….. (can do more things!)



Approach-1. Kernel pointer spraying (KptrSpray)

C-1 ---> **C-2. How to put sensitive kernel pointer on the leak offset?** ---> C-3

How do we exploit eBPF to do spraying?



```
struct bpf_insn bpf_prog[] = {
… // BPF_REG_FP is constant. It doesn't contain real frame pointer value.
BPF_STX_MEM(BPF_DW, BPF_REG_FP, BPF_REG_FP, 0),      // *fp = fp;
…
BPF_STX_MEM(BPF_DW, BPF_REG_FP, BPF_REG_FP, -512), // *(fp-512) = fp;
}
```

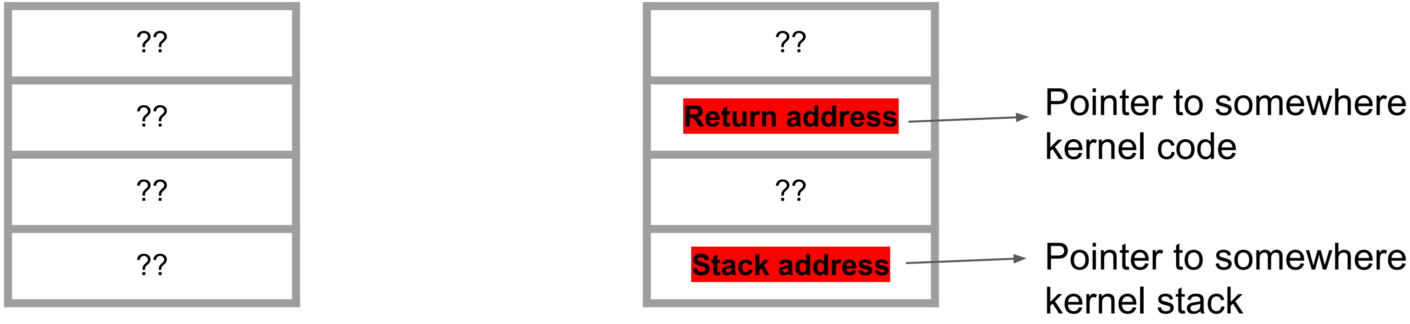
```
..
BPF_STX_MEM(BPF_DW, BPF_REG_FP, BPF_REG_FP, 0),
// 1) Interpreting it.  ⇒ BPF_STX_MEM(BPF_DW,  0xffff0800123000, 0xffff0800123000, 0)
// 2) Running it ⇒ *(0xffff0800123000) = 0xffff0800123000;
// 3) Repeat until it reaches up to the 512 byte! (limitation value of eBPF)
```

Approach-2. Kernel pointer fuzzing (KptrFuzz)

C-1 ---> **C-2. How to put sensitive kernel pointer on the leak offset?** ---> C-3

Kernel Stack
before calling system call A

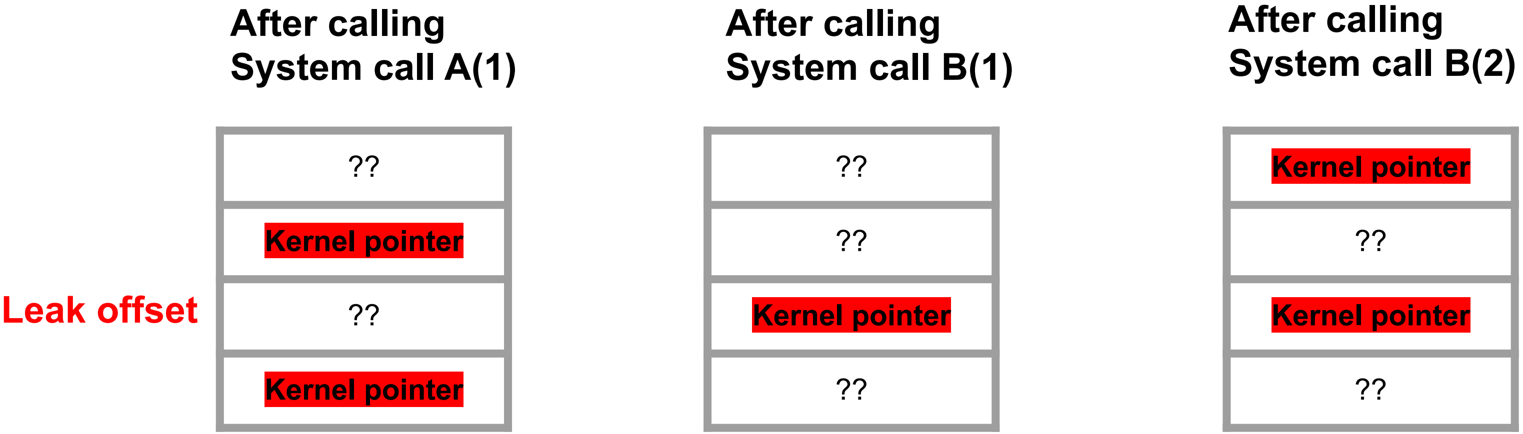
Kernel Stack
after calling system call A



- Return address, kernel stack address will be stored naturally to random kernel stack memory while executing a system call. It means that just calling a system call is helpful for solving this challenge.

Approach-2. Kernel pointer fuzzing (KptrFuzz)

C-1 ---> **C-2. How to put sensitive kernel pointer on the leak offset?** ---> C-3

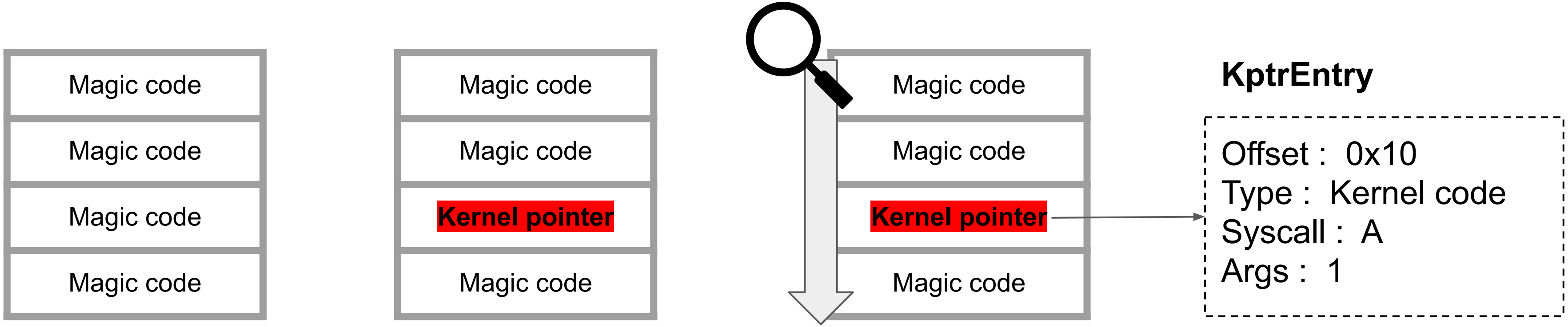


	After calling System call A(1)	After calling System call B(1)	After calling System call B(2)
	??	??	**Kernel pointer**
	Kernel pointer	??	??
Leak offset	??	**Kernel pointer**	**Kernel pointer**
	Kernel pointer	??	??

- Whatever Leak offset is, We can find the system call to put sensitive kernel pointer on the Leak offset with high probability due to a lot of combination of system calls.

Approach-2. KptrFuzz - Implementation

C-1 ---> **C-2. How to put sensitive kernel pointer on the leak offset?** ---> C-3



KptrEntry

Offset : 0x10
Type : Kernel code
Syscall : A
Args : 1

1. Fill kernel stack memory with Magic code.
2. Run a selected system call with selected arguments.
3. Inspect kernel stack memory to find where kernel pointer is located.
4. Recording the context which is called KptrEntry.

Repeat 1~4 on combination of system call as many as possible!!

Approach-2. KptrFuzz - Implementation

C-1 ---> **C-2. How to put sensitive kernel pointer on the leak offset?** ---> C-3

2. Run a selected system call with selected arguments.

- For this step, Either KptrFuzz use their own fuzzing system or use existing Linux fuzzer.

Fuzzer	Kernel Code Coverage (KC)	Kernel Stack Coverage (KS)	Total (KC U KS)
TinySysFuzz	58 %	65 %	75 %
Linux Test Project (LTP)	78 %	80 %	81 %

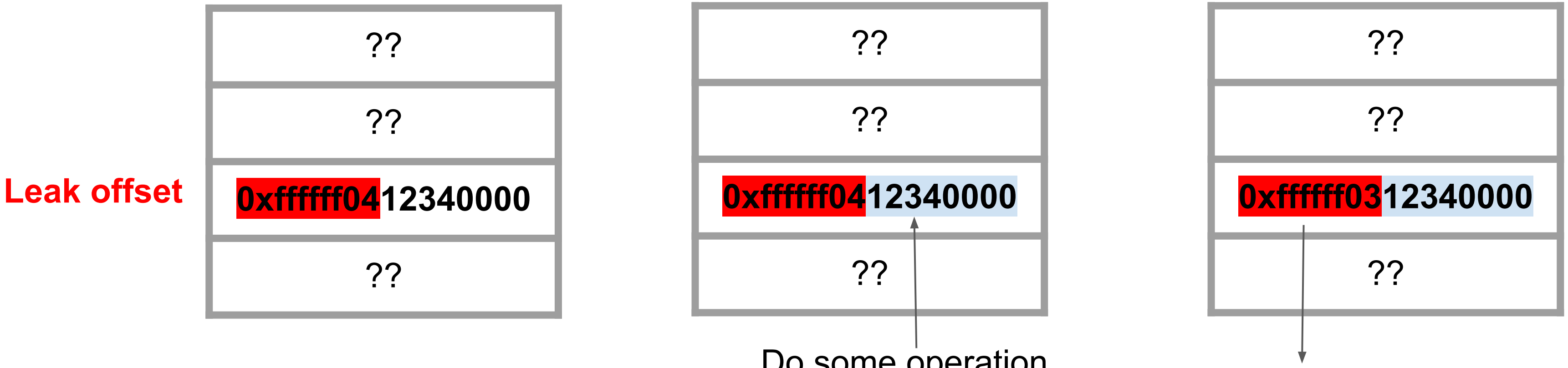
KptrFuzz coverage

- Coverage on 8-byte aligned memory
- Inspection from stack base to 1,260 byte far. (90% of syscall use under 1260 byte)
- Tested on Ubuntu 4.8.0-58-generic kernel

C-3. What if leak size is less than 8 bytes?

Approach-1. Do some operation on unknown bytes

C-1 ---> C-2 ---> **C-3. What If leak size is less than 8 bytes?**



Do some operation

Known bytes are changed!
By using this change,
Infer unknown bytes reversely.

Assumption

- Attacker can do some operation on unknown bytes.

Approach-1. Do some operation on unknown bytes

C-1 ---> C-2 ---> **C-3. What If leak size is less than 8 bytes?**

0xffffff0412340000

Attacker doesn’t know anything.

0xffffff0412340000

Trigger leak!

0xffffff0312340000

Do sub operation! And Trigger leak again!
- 0xffffff0412340000 – 0x0000000012360000
- Attacker knows the unknown bytes is less than 0x12360000

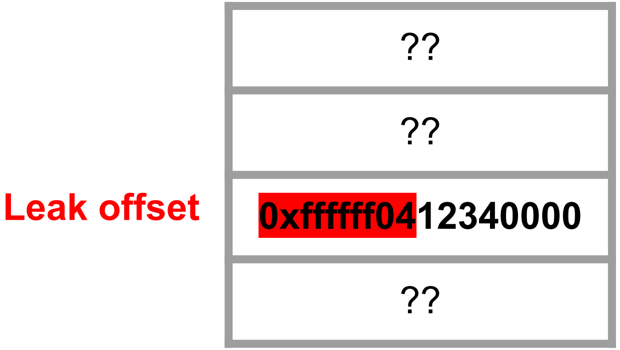
0xffffff0412340000

Do sub operation! And Trigger leak again!
- 0xffffff0412340000 – 0x0000000012300000
- Attacker knows the unknown bytes is greater than 0x12300000
- Attacker knows the unknown bytes is, 0x12300000 ~ 0x12360000.

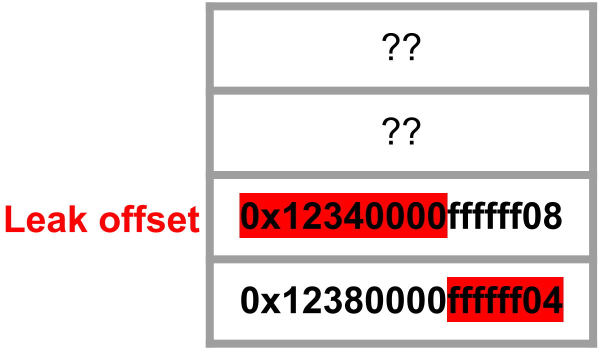
Repeat until get correct answer!

Approach-2. KptrFuzz on N-byte aligned memory

C-1 ---> C-2 ---> **C-3. What If leak size is less than 8 bytes?**



Kernel pointer on 8byte aligned memory



Kernel pointer on 4byte aligned memory

- In the case that Leak size is 4, We can run KptrFuzz multilple times to get full 8-byte kernel pointer.

  1. KptrFuzz on 8byte aligned memory leaks high 4byte of kernel pointer.
  2. KptrFuzz on 4byte aligned memory leaks remaining low 4byte of kernel pointer.

Approach-2. KptrFuzz on N-byte aligned memory

C-1 ---> C-2 ---> **C-3. What If leak size is less than 8 bytes?**

Fuzzer	**Coverage (8-byte aligned)**	**Coverage (4-byte aligned)**
TinySysFuzz	75 %	0 %
Linux Test Project (LTP)	81 %	0 %

KptrFuzz coverage

- Unfortunately, KptrEntry which is aligned smaller than 8-byte memory is not exist in kernel stack. So… Even if this approach is possible theoretically, but couldn't applied to real-world Linux kernel.

** When sensitive kernel pointer has been overwritten unintentionally. (Failed to solve)

In-depth of CVE-2016-5244

Vulnerability

```
void rds_inc_info_copy(struct rds_incoming *inc,
               struct rds_info_iterator *iter,
               __be32 saddr, __be32 daddr, int flip)
{
    struct rds_info_message minfo;

    minfo.seq = be64_to_cpu(inc->i_hdr.h_sequence);
    minfo.len = be32_to_cpu(inc->i_hdr.h_len);

    if (flip) {
        minfo.laddr = daddr;
        minfo.faddr = saddr;
        minfo.lport = inc->i_hdr.h_dport;
        minfo.fport = inc->i_hdr.h_sport;
    } else {
        minfo.laddr = saddr;
        minfo.faddr = daddr;
        minfo.lport = inc->i_hdr.h_sport;
        minfo.fport = inc->i_hdr.h_dport;
    }

    rds_info_copy(iter, &minfo, sizeof(minfo));
}
```

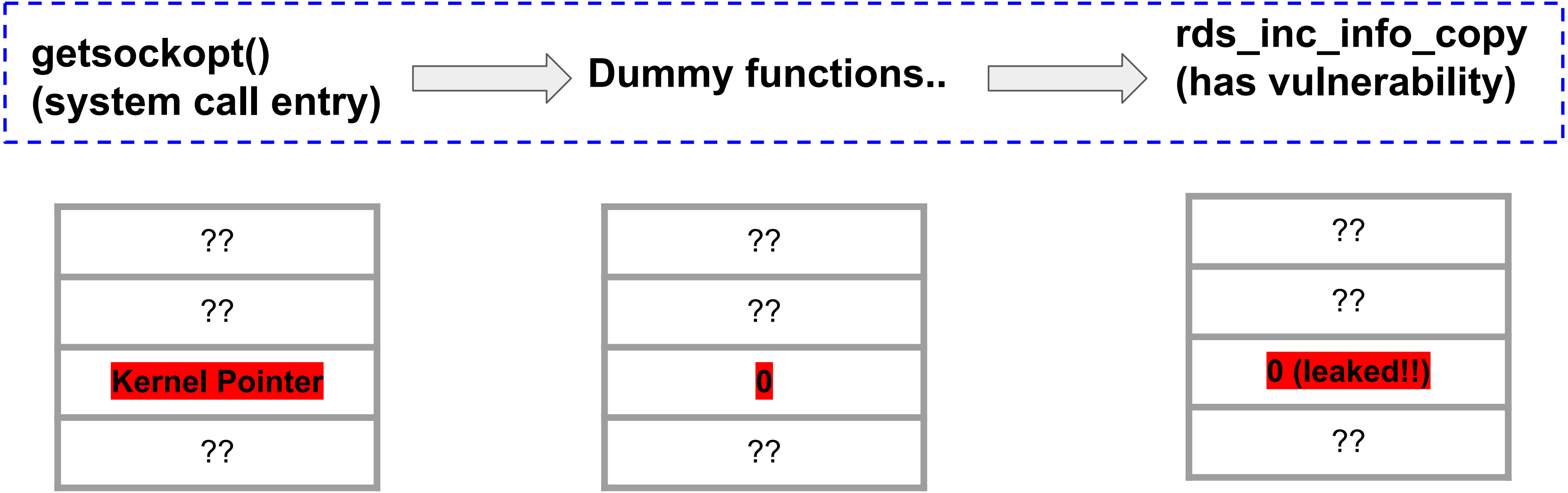
It has 7 fields.
But, It only initializes 6 fields.
minfo.flags will be uninitialized.

Leak to user-space

In-depth of CVE-2016-5244

Problem

Control Flow Path from system-call-entry to vulnerability



getsockopt()
(system call entry)
→ **Dummy functions..** →
rds_inc_info_copy
(has vulnerability)

??	??	??
??	??	??
Kernel Pointer	**0**	**0 (leaked!!)**
??	??	??

Exploitations on real-world vulnerabilities

Exploitations on real-world vulnerabilities

CVE	Leak size	C-1	C-2	C-3	Result	Code
CVE-2016-4486	4 byte	Footprinting	KptrSpray	Do some operation...	8 byte pointer to kernel stack	[1]
CVE-2018-11508	4 byte	Footprinting	KptrFuzz	-	4 byte pointer to kernel code (enough to bypass KASLR)	[2]
CVE-2016-4569	4 byte	Footprinting	KptrFuzz	-	4 byte pointer to kernel code	-
Not assigned [3]	4 byte	Footprinting	KptrFuzz	-	4 byte pointer to kernel code	-

- C-1 : How to calculate leak offset?
- C-2 : How to put sensitive kernel pointer on the leak offset?
- C-3 : What if leak size is less than 8 bytes?
- [1] https://www.exploit-db.com/exploits/46006
- [2] https://www.exploit-db.com/exploits/46208
- [3]] https://github.com/torvalds/linux/commit/7c8a61d9ee

Demo-1

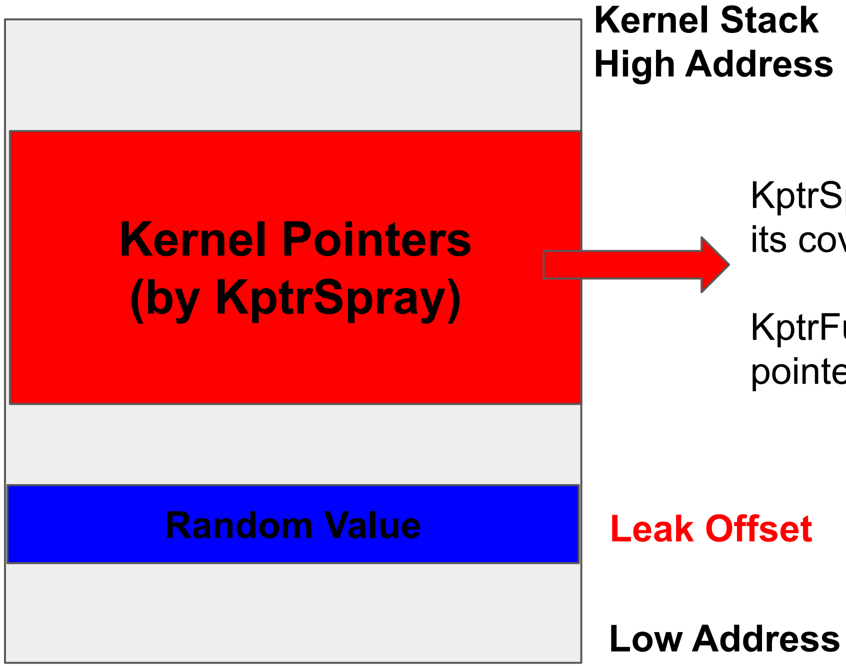
- Exploiting CVE-2018-11508
- Using Footprinting, KptrFuzz
- Goal : Bypassing KASLR
- Leak Size : 4 bytes

Demo-2

- Exploiting CVE-2016-4486
- Using KptrSpray, Do some operation…
- Goal : Get 8-byte kernel stack address
- Leak Size : 4 bytes

Exploiting CVE-2016-4486

One More Problem Here!



KptrSpray couldn’t clobber value at Leak Offset since its coverage limitation…

KptrFuzz also couldn’t find a system call to put kernel pointer onto the Leak Offset...

What should we do?

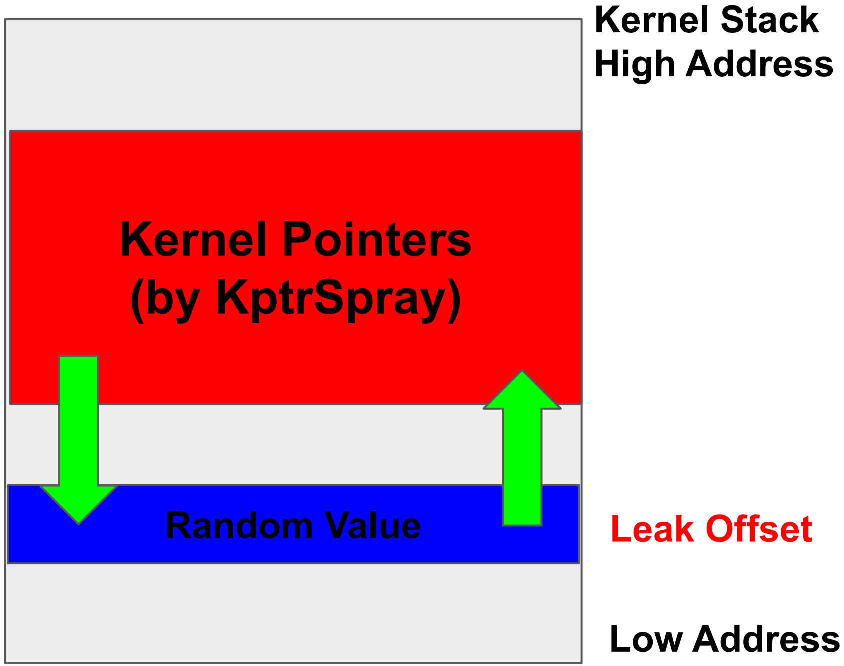
Exploiting CVE-2016-4486

Possible Solution

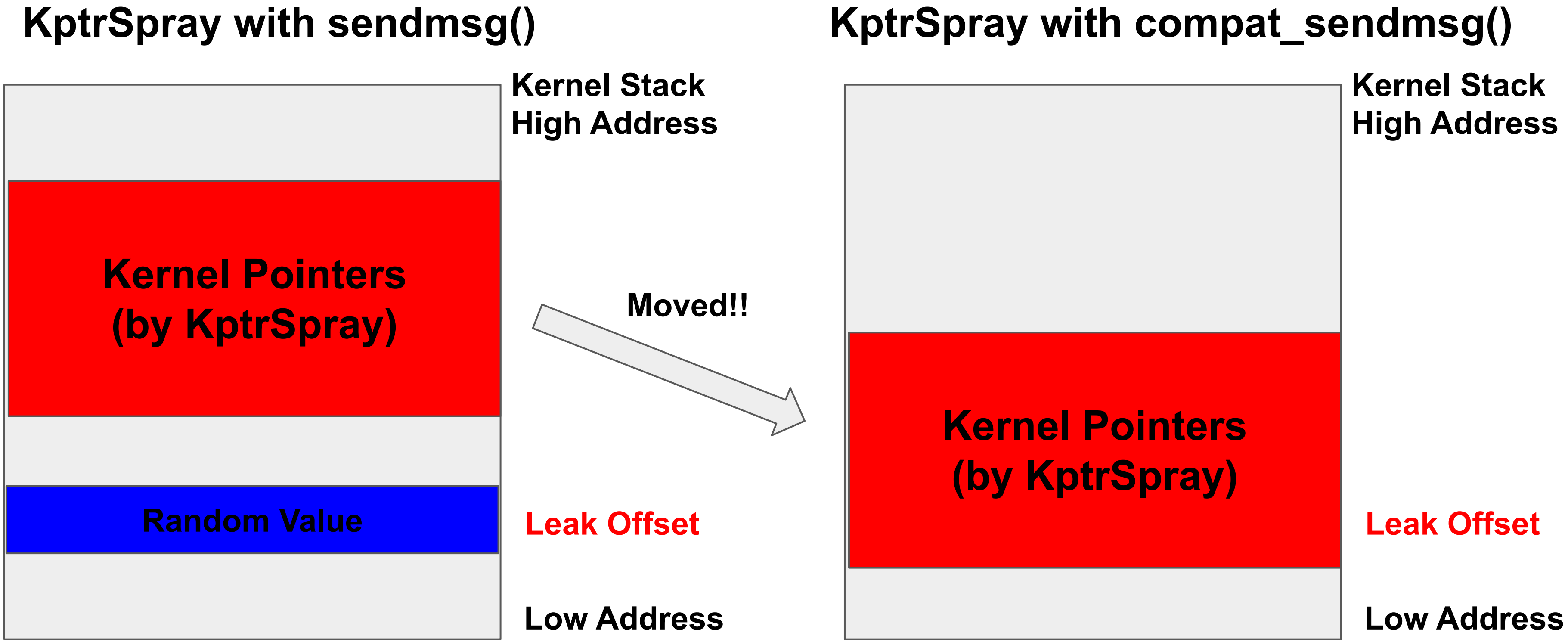
Either
- Moving KptrSpray zone downward to clobber Leak Offset.
- Or Moving LeakOffset up to the KptrSpray zone.

Why is it possible??

- Stack Address is not absolute address. It's depending on the control flow path.
- If we try all possible control flow paths to trigger KptrSpray or Vulnerability, we could find a case to solve this challenge!



Exploiting CVE-2016-4486



Demo-2

- Exploiting CVE-2016-4486
- Using KptrSpray, Do some operation…
- Goal : Get 8-byte kernel stack address
- Leak Size : 4 bytes

Mitigations

Runtime prevention

1. **STACKLEAK**

- Implemented as GCC Plugin.
- From grsecurity/PaX team.
- Integrated into Linux kernel upstream since 2018.09 (v4.20 - latest!!)
- Not default option of Linux kernel.
- Zeroing out stack when syscall returns, The zeroing eliminates all sensitive information inside stack, So that attacker can’t get anything through exploitation.

Bug finding (Static method)

1. **UniSan**

- From georgia tech as an academic paper. (ACM CCS 2016)
- Presents **Static analysis tool** for finding information leak caused by uninitialized use from both Stack and Heap.
- https://github.com/sslab-gatech/unisan (OpenSource)

Bug finding (Dynamic method)

1. **KMSAN (Kernel Memory Sanitizer)**

- From google as one of memory sanitizer project.
- Detector of uninitialized use for the Linux kernel. (currently in development)
- Presents runtime detection using both Syzkaller and KMSAN-applied kernel.
- :- 24-hours running on Syzbot.

Thank you!

Q & A