SOSCON
Attack and Defense on Linux kernel

SAMSUNG Research | Security Team | Jinbum Park

2018.10.18

SOSCON 2018

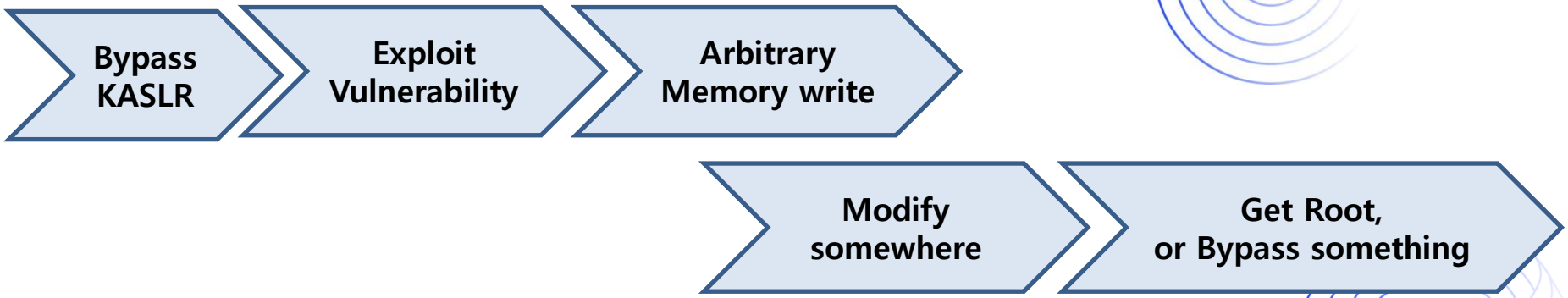
SAMSUNG OPEN SOURCE CONFERENCE 2018

Contents

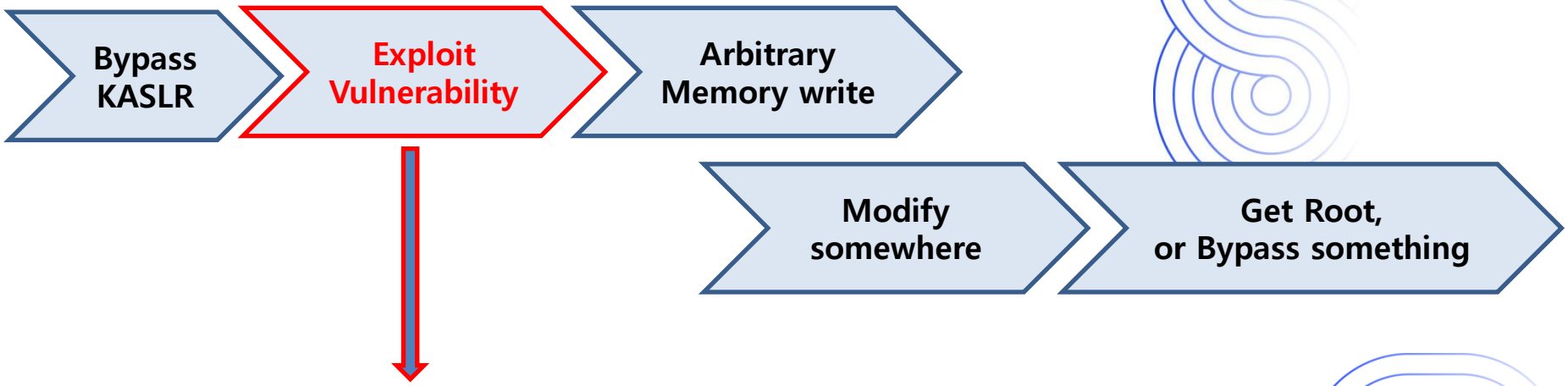
Full steps of attack on Linux kernel

SOSCON 2018
SAMSUNG OPEN SOURCE CONFERENCE 2018

Full steps of attack on Linux kernel

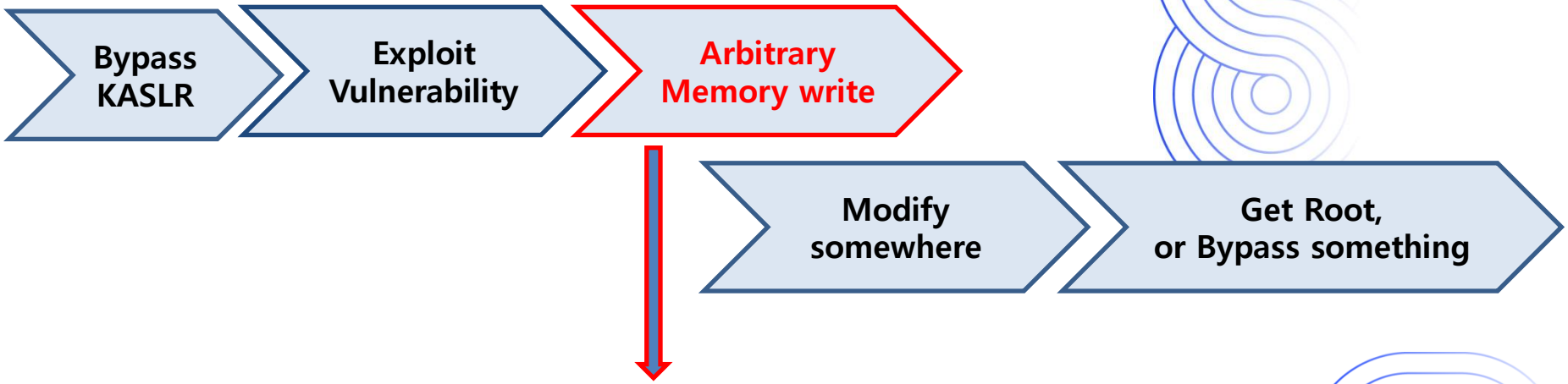


Full steps of attack on Linux kernel



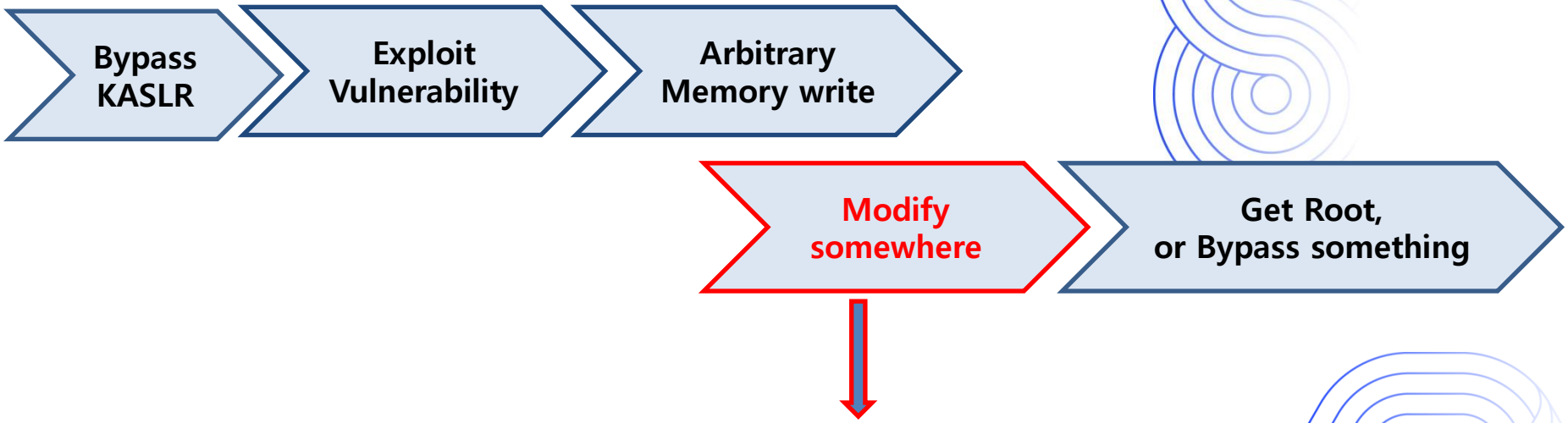
- Exploit Linux kernel vulnerabilities.
- By exploiting them, Attacker can
  - Modify control flow,
  - Do arbitrary memory write.

Full steps of attack on Linux kernel



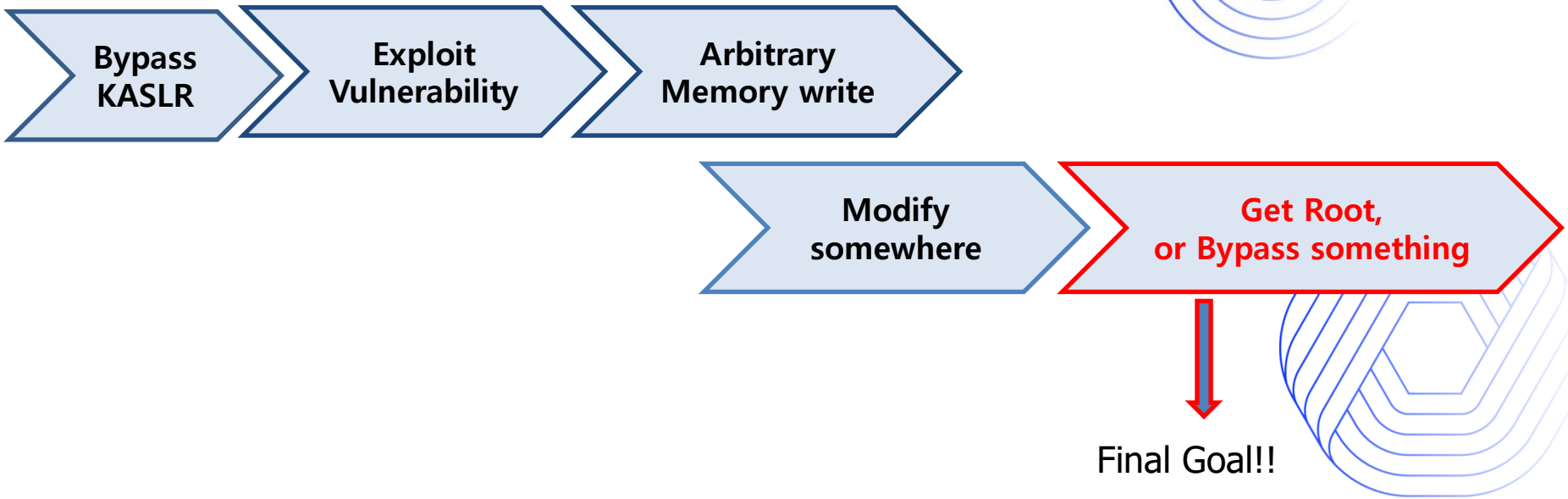
- It means ability to modify
  - Limited RW Linux kernel memory, or Any RW Linux kernel memory.
- Can attacker write any values they want?
  - Depends on previous step "Exploit Vulnerability"

Full steps of attack on Linux kernel



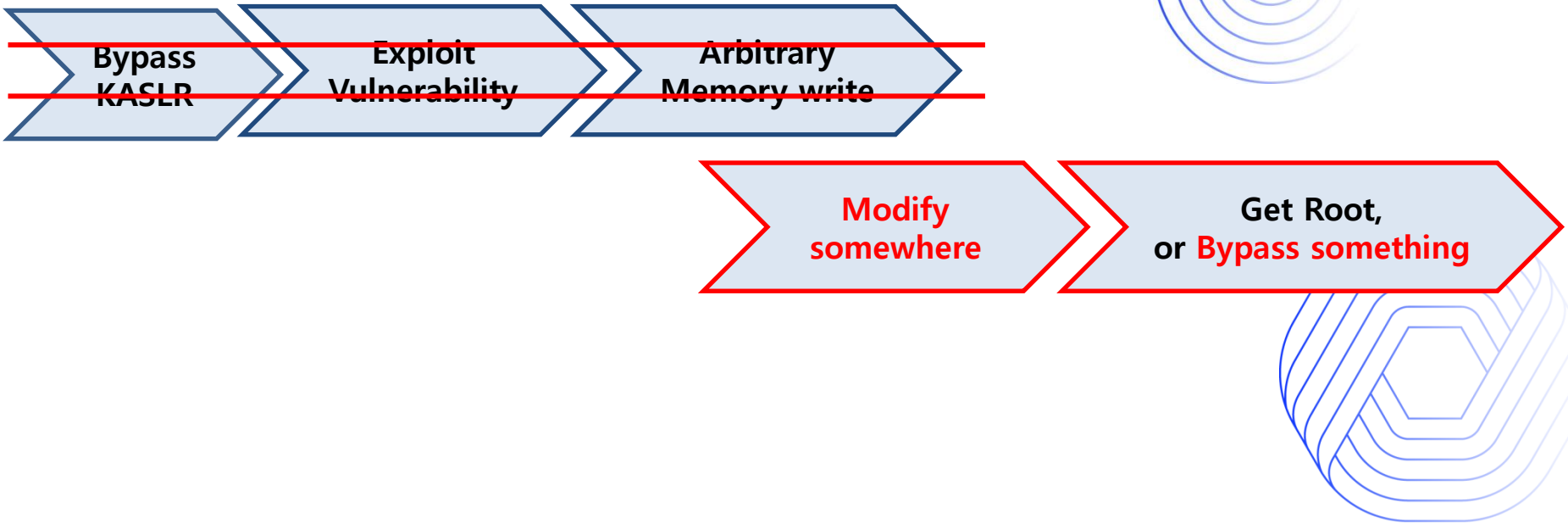
- Which data is the most attractive for attacker?
  - Function Pointer, Flags which are used for security checking.
- Attacker can get root by modifying function pointer.
- Attacker can bypass security mechanism by modifying some flags.

Full steps of attack on Linux kernel



Attack-1 :
Modify sensitive RW data

Modify sensitive RW data



Modify sensitive RW data

Sensitive RW data : Function Pointer

Why function pointer is critical?

Let's look at Linux kernel 3.10.

```
static struct security_operations selinux_ops = {
    .name =                 "selinux",
```

```
.secmark_relabel_packet =    selinux_secmark_relabel_packet,
.secmark_refcount_inc =      selinux_secmark_refcount_inc,
.secmark_refcount_dec =      selinux_secmark_refcount_dec,
```



```
void (*secmark_refcount_inc) (void);
void (*secmark_refcount_dec) (void);
```

- If attacker can modify function pointer - .secmark_refcount_inc,
  What can attacker do?

Modify sensitive RW data

Sensitive RW data : Function Pointer

```
static struct security_operations selinux_ops = {
    .name =                 "selinux",

.secmark_relabel_packet =   selinux_secmark_relabel_packet,
.secmark_refcount_inc =     selinux_secmark_refcount_inc,
.secmark_refcount_dec =     selinux_secmark_refcount_dec,
```

```
void (*secmark_refcount_inc) (void);
void (*secmark_refcount_dec) (void);
```

Normal

```
static void selinux_secmark_refcount_inc(void)
{
```

Malicious!!

```
void reset_security_ops(void)
{
    security_ops = &default_security_ops;
}
```

Possible??

```
int security_capget(struct task_struct *target,
                    kernel_cap_t *effective,
                    kernel_cap_t *inheritable,
                    kernel_cap_t *permitted)
```

- Attacker can call other security-critical function which has same function type.
- "reset_security_ops()" disables Linux security module such as Smack, SELinux, …

  So that, Attacker can bypass Linux security module!!

Modify sensitive RW data

Sensitive RW data : Flags which are used for security checking

Let's look at Linux kernel 3.10.

```
static struct sidtab sidtab;
struct policydb policydb;
int ss_initialized;
```

Flag to represent whether SELinux is initialized or not.

```
int security_load_policy(void *data, size_t len)
{
if (!ss_initialized) {
    avtab_cache_init();
    rc = policydb_read(&policydb, fp);
```

Used for security checking!

If attacker can set this to 0,
Reinitializing SELinux policy is possible!!
And other operations too!!

Modify sensitive RW data

Sensitive RW data : Flags which are used for security checking

```
static struct sidtab sidtab;
struct policydb policydb;
int ss_initialized;
```

Flag to represent whether SELinux is initialized or not.

- **Defeating Samsung KNOX with zero privilege, Di shen, Blackhat USA 2017**

➔ This was a real world attack to hack Galaxy S7 edge!!

Defense-1 : ro-after-init

ro-after-init

Read only after initialization

- What is a key insight inside ro-after-init?

  - A lot of RW data are used to be written only one time.
  - When?? ➔ Kernel Initialization time!!
  - Then?? ➔ The RW data can be marked as read-only after initialization!
  - It reduces a lot of attack surface with no performance overhead!!

ro-after-init

Read only after initialization

- How to apply ro-after-init?

```
static struct genl_family family __ro_after_init = {
    .name       = TASKSTATS_GENL_NAME,
    .version    = TASKSTATS_GENL_VERSION,
    .maxattr    = TASKSTATS_CMD_ATTR_MAX,
    .module     = THIS_MODULE,
    .ops        = taskstats_ops,
    .n_ops      = ARRAY_SIZE(taskstats_ops),
};
```

- Just add keyword "_ro_after_init" to variables which you want to protect.
- Limitation : Developer should know which variables can be marked as ro-after-init.
  Automatic process for marking them has not been appeared yet.

ro-after-init

Read only after initialization

- Real-world cases for protecting function pointers

```
static struct security_hook_list smack_hooks[] = {
    LSM_HOOK_INIT(ptrace_access_check, smack_ptrace_access_check),
    LSM_HOOK_INIT(ptrace_traceme, smack_ptrace_traceme),
    LSM_HOOK_INIT(syslog, smack_syslog),
```

Linux kernel 4.8

```
static struct security_hook_list smack_hooks[] __lsm_ro_after_init = {
    LSM_HOOK_INIT(ptrace_access_check, smack_ptrace_access_check),
    LSM_HOOK_INIT(ptrace_traceme, smack_ptrace_traceme),
    LSM_HOOK_INIT(syslog, smack_syslog),
```

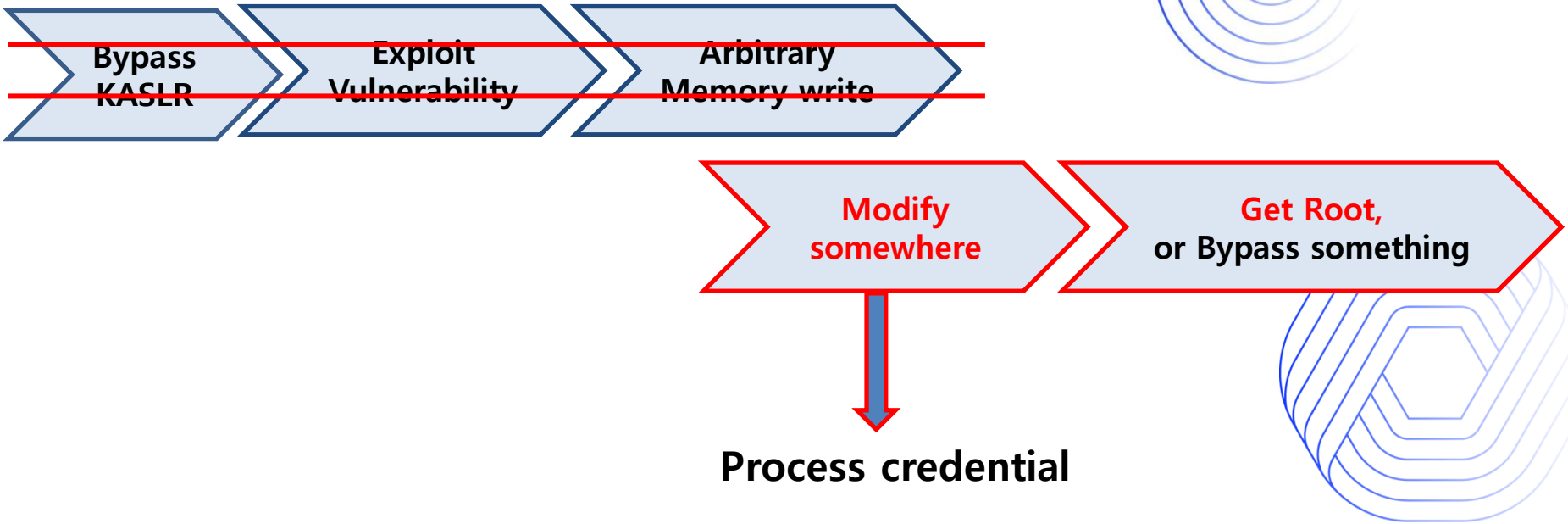
Linux kernel 4.12

Summary

Reduce attack surface as much as possible!!

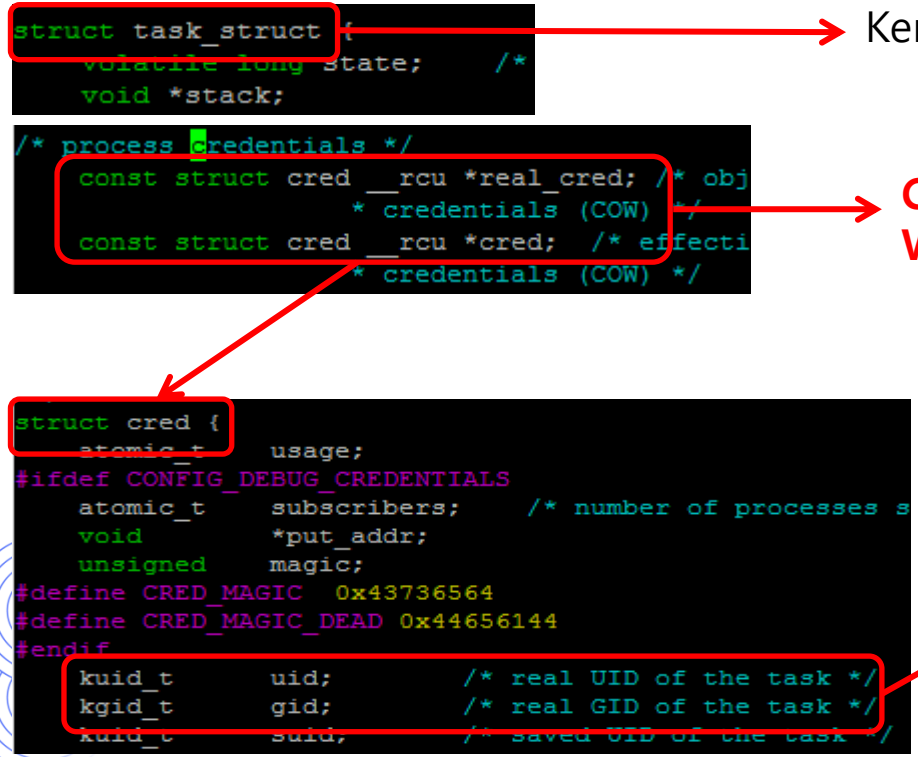
Attack-2 :
Modify process credential

Modify process credential



Modify process credential

cred



Kernel structure to represent one process

Credential for this process.
We will modify this!

Credential is tightly related to permission of process!!

Modify process credential

Type1 : Function calls to modify cred for root

- Attacker executes below two function calls. (kernel function)

```
commit_creds(prepare_kernel_cred(0));
```

Apply the new cred to current process

Make a new cred for root (uid=0)

```
struct task_struct {
    volatile long state;    /*
    void *stack;
/* process credentials */
    const struct cred __rcu *real_cred; /* obj
                            * credentials (COW) */
    const struct cred __rcu *cred;  /* effecti
                            * credentials (COW) */
```

- These function calls makes attacker to get root!!
- A lot of real-world attacks use this technique,
- CVE-2016-0728, …

Modify process credential

Type2 : Reuse init_cred

```
struct task_struct {
    volatile long state;    /*
    void *stack;
/* process credentials */
    const struct cred __rcu *real_cred; /* obj
                     * credentials (COW) */
    const struct cred __rcu *cred;  /* effecti
                     * credentials (COW) */
```

Original cred for user permission

init_cred for root permission

```
/*
 * The initial credentials for the initial task
 */
struct cred init_cred = {
    .usage          = ATOMIC_INIT(4),
#ifdef CONFIG_DEBUG_CREDENTIALS
    .subscribers        = ATOMIC_INIT(2),
    .magic          = CRED_MAGIC,
#endif
    .uid            = GLOBAL_ROOT_UID,
    .gid            = GLOBAL_ROOT_GID,
```

Modify process credential

Type3 : Modify cred itself

```
struct task_struct {
    volatile long state;    /*
    void *stack;
```

```
/* process credentials */
    const struct cred __rcu *real_cred; /* obj
                    * credentials (COW) */
    const struct cred __rcu *cred;  /* effecti
                    * credentials (COW) */
```

```
struct cred {
    atomic_t    usage;
#ifdef CONFIG_DEBUG_CREDENTIALS
    atomic_t    subscribers;    /* number of processes s
    void        *put_addr;
    unsigned    magic;
#define CRED_MAGIC  0x43736564
#define CRED_MAGIC_DEAD 0x44656144
#endif
    kuid_t      uid;        /* real UID of the task */
    kgid_t      gid;        /* real GID of the task */
    kuid_t      suid;       /* saved UID of the task */
```

Modify these directly!!

Defense-2 : PrivWatcher

PrivWatcher

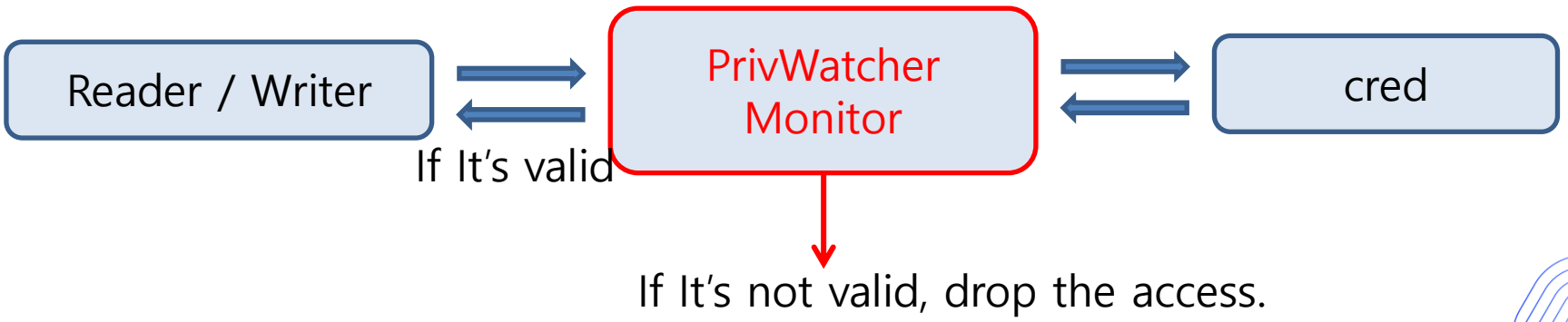
Paper

- **PrivWatcher: Non-bypassable Monitoring and Protection of Process Credentials from Memory Corruption Attacks,**
  AsiaCCS 2017, Samsung Research America

➔ This is a paper proposed by Samsung Research America!!
➔ This is not merged in Linux kernel mainline.
➔ Is this merged in Linux kernel for Galaxy??

PrivWatcher

Simple principle for defense



- Attack Type1 : Function calls to modify cred
- Attack Type2 : Reuse init_cred
- Attack Type3 : Manipulate cred itself

➔ PrivWatcher can prevent all attack types!! Prevent privilege escalation through cred.

PrivWatcher

Is this merged in Linux kernel for Galaxy?

```
#ifdef CONFIG_RKP_KDP
    atomic_t *use_cnt;
    struct task_struct *bp_task;
    void *bp_pgd;
    unsigned long long type;
#endif /*CONFIG_RKP_KDP*/
```

```
/* Main function to verify cred security context of a process */
int security_integrity_current(void)
{
    if ( rkp_cred_enable &&
        (rkp_is_valid_cred_sp((u64)current_cred(),(u64)current_cred()->security)||
        cmp_sec_integrity(current_cred(),current->mm)||
        cmp_ns_integrity())) {
        rkp_print_debug();
        panic("RKP CRED PROTECTION VIOLATION\n");
    }
    return 0;
}
```

Galaxy Note9 Kernel Code

- Not same solution to PrivWatcher.
  But, There is a similar solution
  in after Galaxy S7.

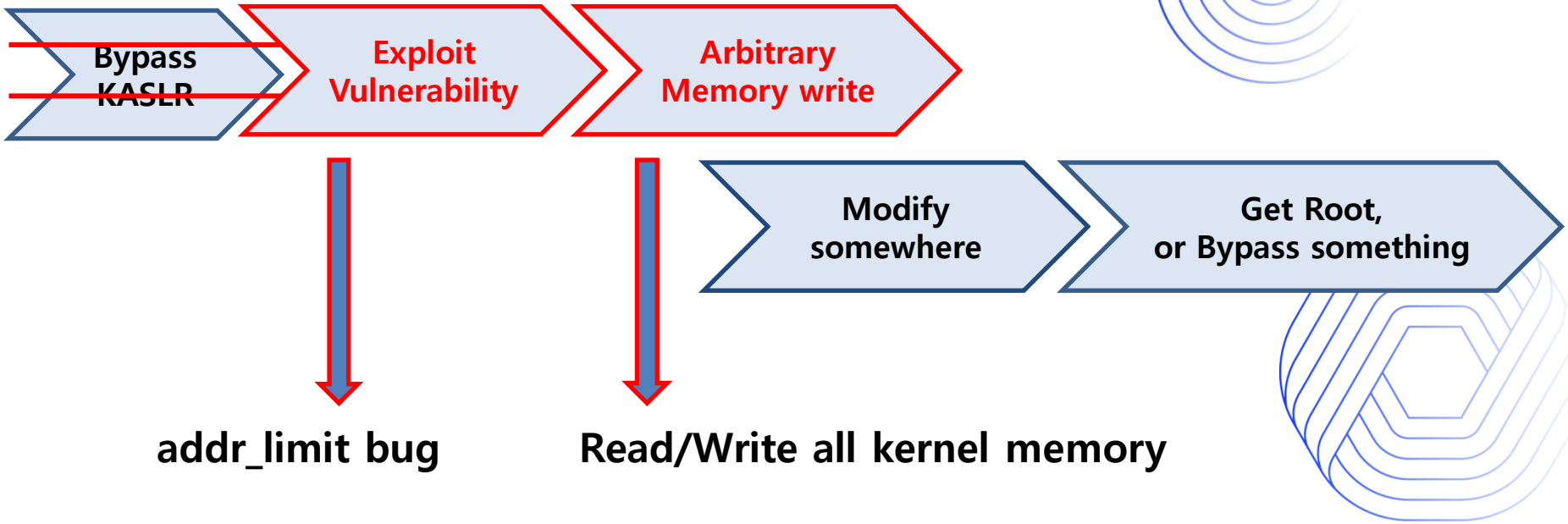
Modify process credential

Summary

You can add your security solution into your product!

Attack-3 :
addr_limit bug

addr_limit bug



addr_limit bug

What is addr_limit?

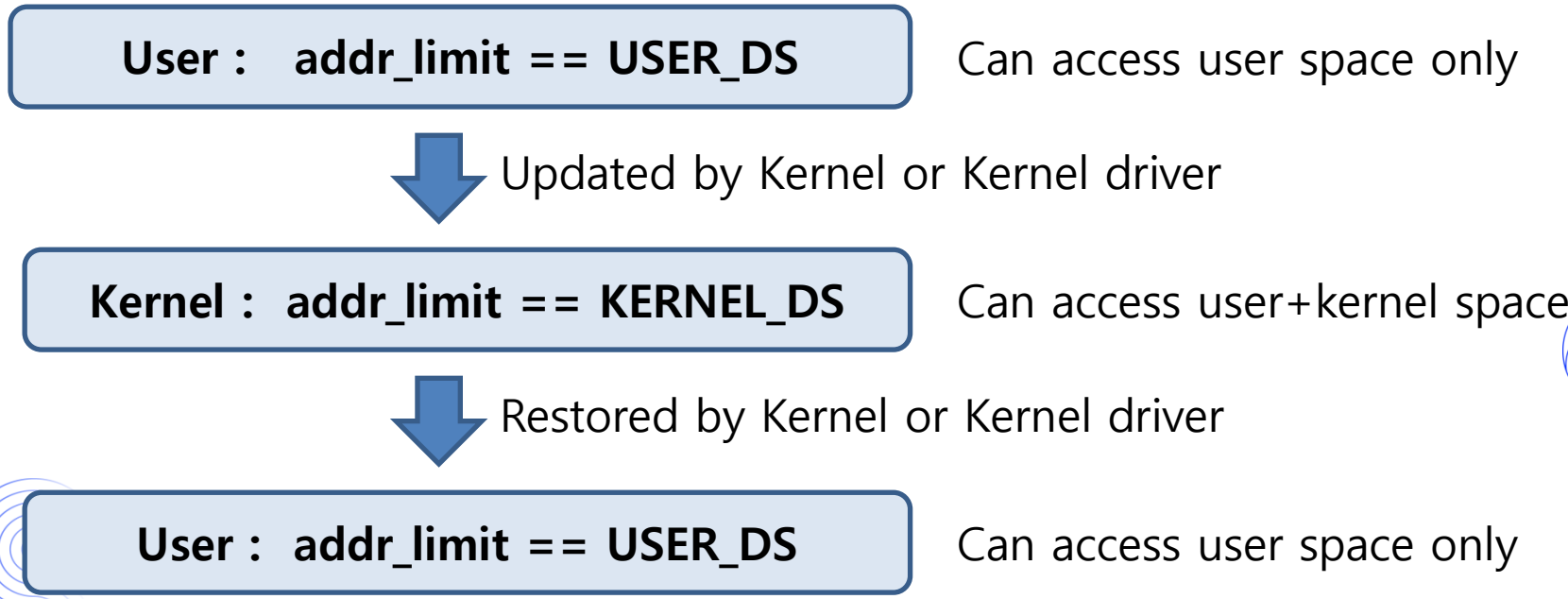
- Look at "struct thread_info" which is generated per process.
- It's different per CPU type. Below one is for arm64.

```
/*
 * low level task data that entry.S needs immediate access to.
 * __switch_to() assumes cpu_context follows immediately after cpu_domain.
 */
struct thread_info {
    unsigned long       flags;      /* low level flags */
    mm_segment_t        addr_limit; /* address limit */
    struct task_struct  *task;      /* main task structure */
    int         preempt_count;  /* 0 => preemptable, <0 => bug */
    int         cpu;        /* cpu */
};
```

- addr_limit have a role like partition between user and kernel space.

addr_limit bug

Normal state-flow of addr_limit



addr_limit bug

Mistaken state-flow of addr_limit (mistakes from developer)

User : addr_limit == USER_DS — Can access user space only

Kernel : addr_limit == KERNEL_DS — Can access user+kernel space

Miss restore!! (human error)

User : addr_limit == KERNEL_DS

Can access user+kernel space!!
Read/Write all Kernel memory!!

addr_limit bug

Real-world vulnerability

```
int _write_log(char *filename, char *data)
{
  struct file *file;

if (f54_window_crack || f54_window_crack_check_mode == 0) {
  mm_segment_t old_fs = get_fs();
  set_fs(KERNEL_DS);
  flags = O_WRONLY | O_CREAT;

if (filename) {
  file = filp_open(filename, flags, 0666);
  sys_chmod(filename, 0666);
} else {
  TOUCH_E("%s : filename is NULL, can not open FILE\n",
    __func__);
  return -1;
}
```

addr_limit == KERNEL_DS

Not restored!!

- This is one of real-world vulnerabilities, which in LG G4 touch screen driver in Android.

addr_limit bug

How can modify kernel memory actually??

```
memcpy(kernel_addr,  buf,  len);
```

< User >

User : addr_limit == KERNEL_DS

- Then, Can an attacker modify kernel memory like above?? (after addr_limit bug)
  Definitely No…

- How to modify??

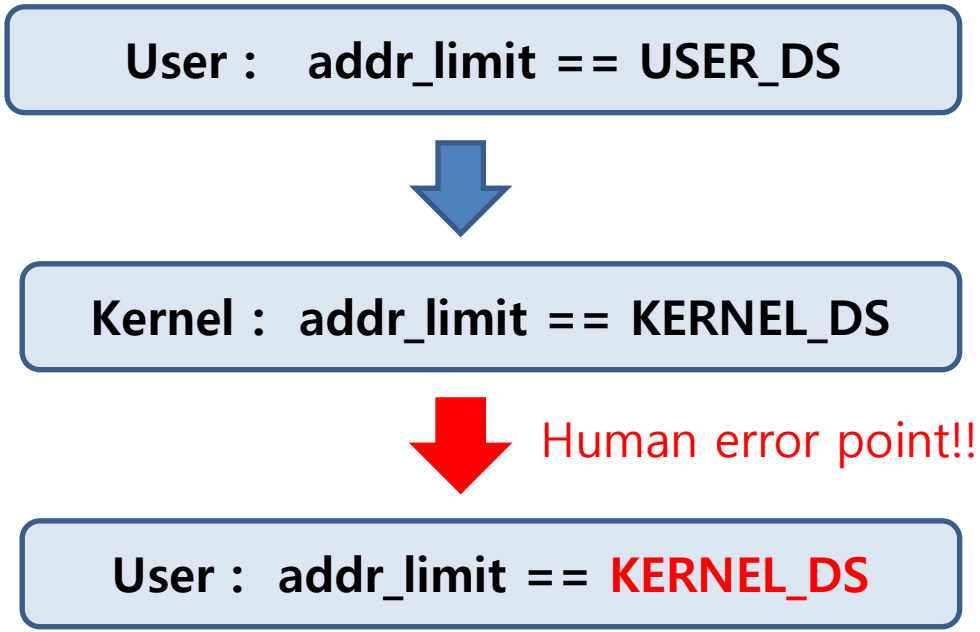
  - Exploiting pipe subsystem (http://blog.daum.net/tlos6733/184)

Defense-3 :
Add checking for addr_limit

Add checking for addr_limit

What is the most critical problem for handling addr_limit?

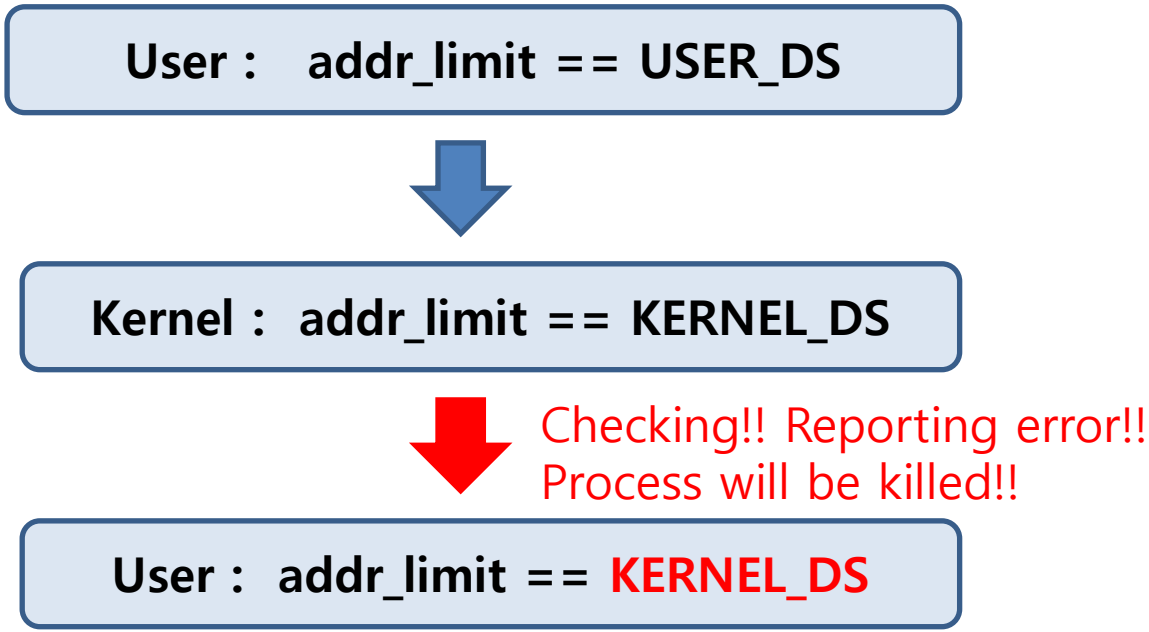
- Possibility of human error!!



Add checking for addr_limit

Solution

- Enforce security-checking when returned from Kernel to User.



Add checking for addr_limit

Solution

```
static inline void set_fs(mm_segment_t fs)
{
    current_thread_info()->addr_limit = fs;

    /* On user-mode return, check fs is correct */
    set_thread_flag(TIF_FSCHECK);
```

```
/*
 * Called before coming back to user-mode. Returning to user-mode with an
 * address limit different than USER_DS can allow to overwrite kernel memory.
 */
static inline void addr_limit_user_check(void)
{
#ifdef TIF_FSCHECK
    if (!test_thread_flag(TIF_FSCHECK))
        return;
#endif

    if (CHECK_DATA_CORRUPTION(!segment_eq(get_fs(), USER_DS),
                "Invalid address limit on user-mode return"))
        force_sig(SIGKILL, current);

#ifdef TIF_FSCHECK
    clear_thread_flag(TIF_FSCHECK);
#endif
}
```

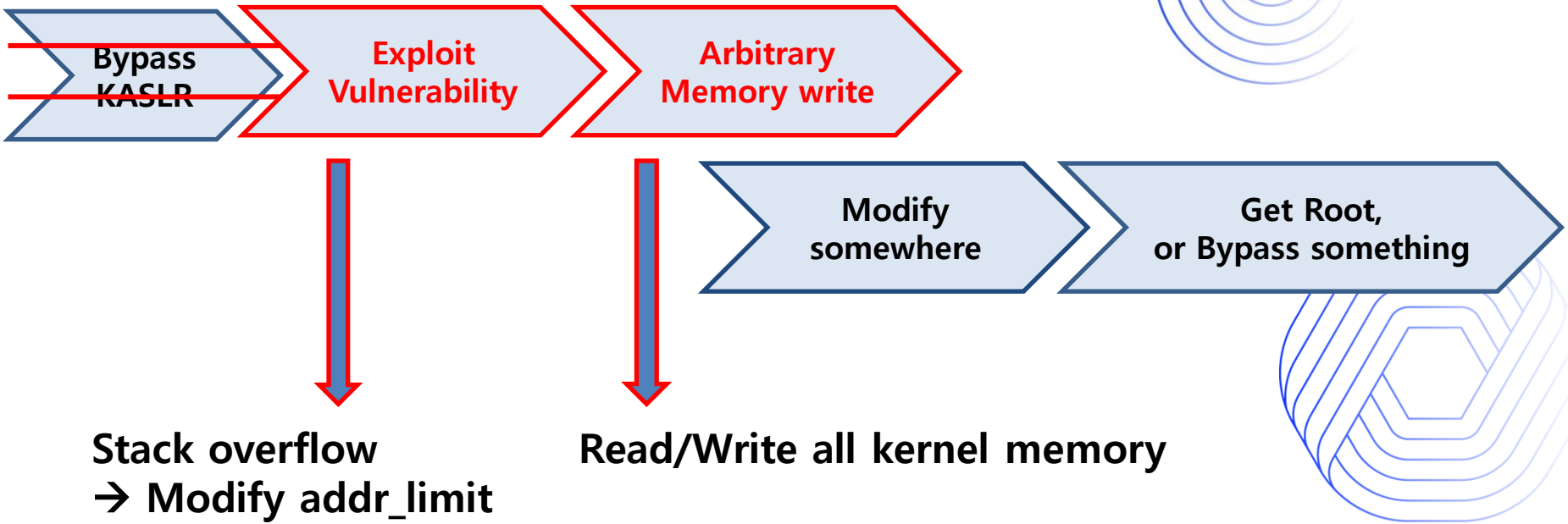
Add checking for addr_limit

Summary

Enforce security checking to eliminate human errors!!

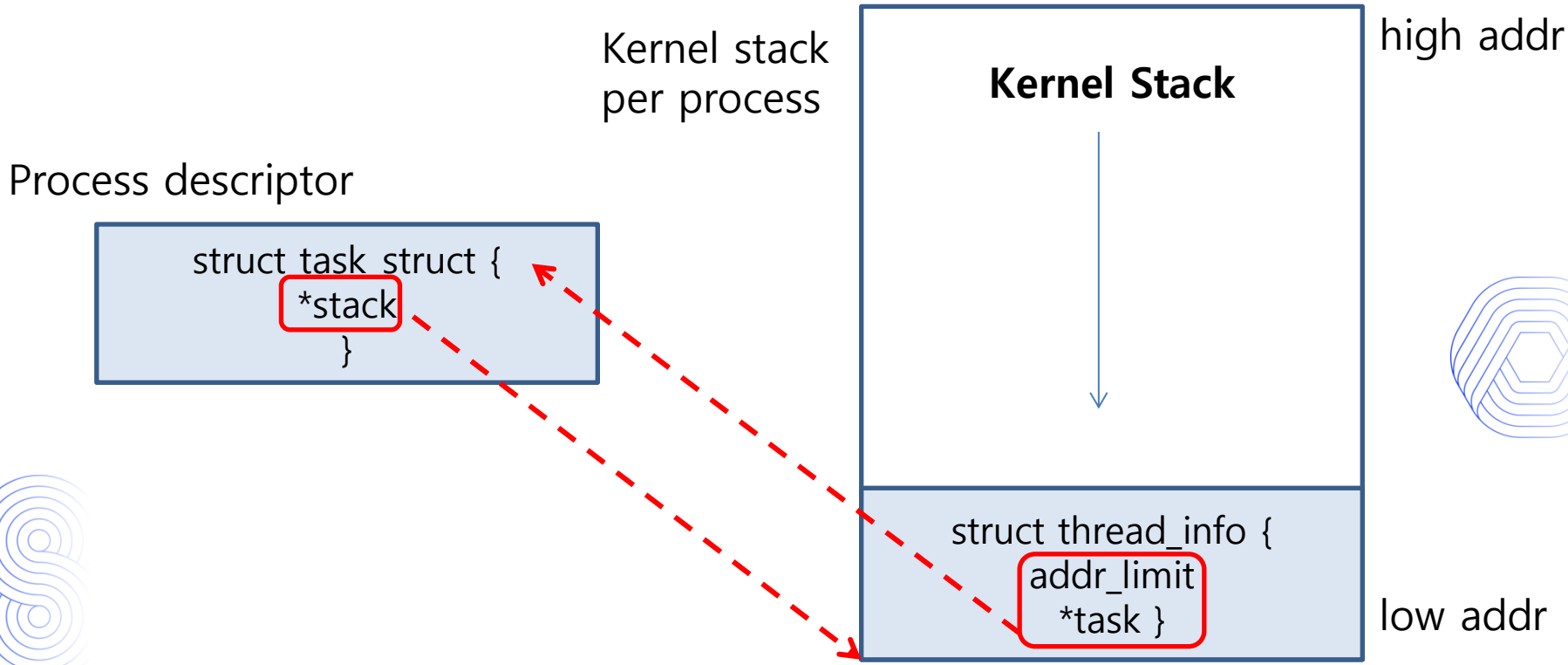
Attack-4 :
Modify addr_limit
via stack-based attack

Modify addr_limit via stack-based attack



Modify addr_limit via stack-based attack

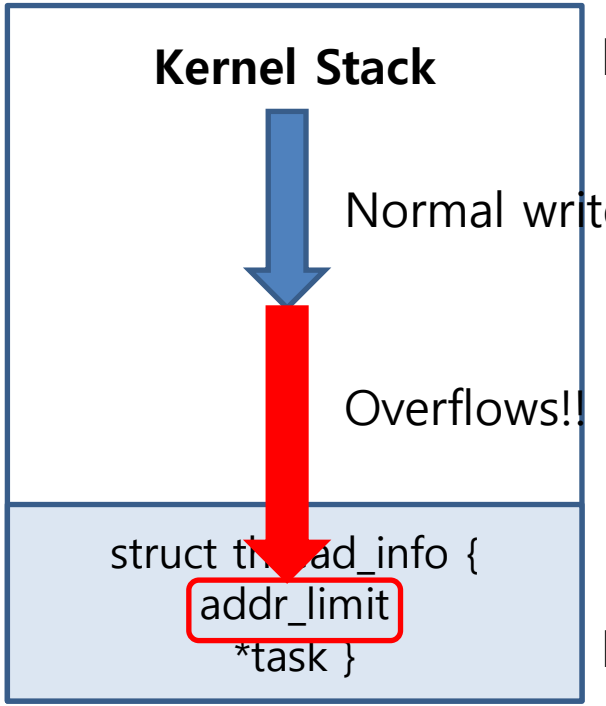
Where "addr_limit" be stored? In kernel stack!!



Modify addr_limit via stack-based attack

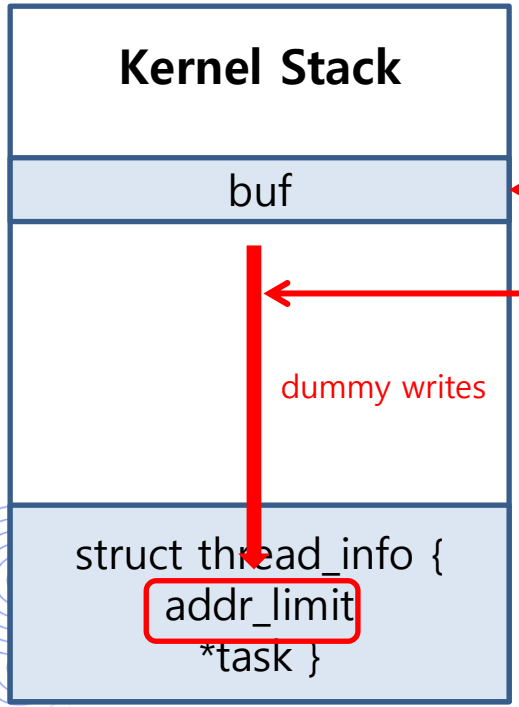
How about trying stack overflow attack as a classic?

Kernel stack per process



Modify addr_limit via stack-based attack

Stack overflow – Type 1: classic buffer overflow



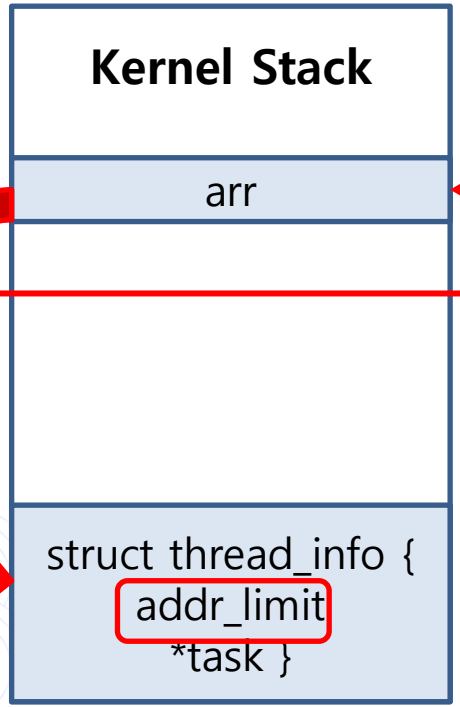
Vulnerability

```
int vul_func(char *arg, unsigned int len)
{
    char buf[64];
    ....
    memcpy(buf, arg, len);
    ....
}
```

Attack succeed? depends on vulnerability.
In some cases, Process may be crashed because of dummy writes.

Modify addr_limit via stack-based attack

Stack overflow – Type 2: out-of-bound index



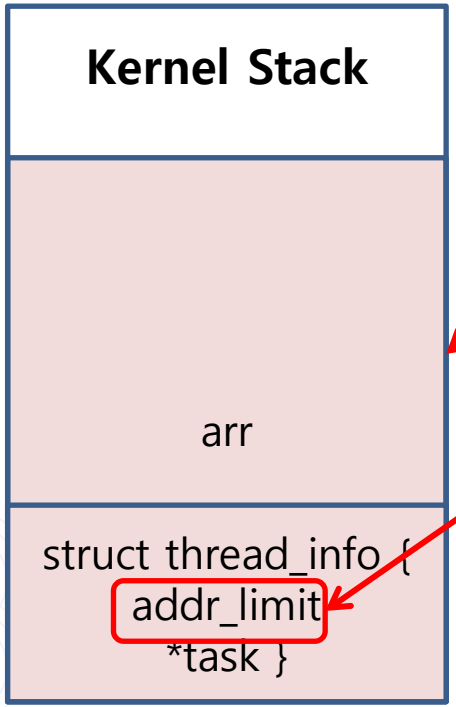
Vulnerability

```
int vul_func(int idx, int val)
{
    int arr[64];
    ....
    arr[idx] = val;
    ....
}
```

Attack succeed? Yap! It seems be possible!!
But,, Is it in real world?? May be no..

Modify addr_limit via stack-based attack

Stack overflow – Type 3: VLA (Variable Length Array)



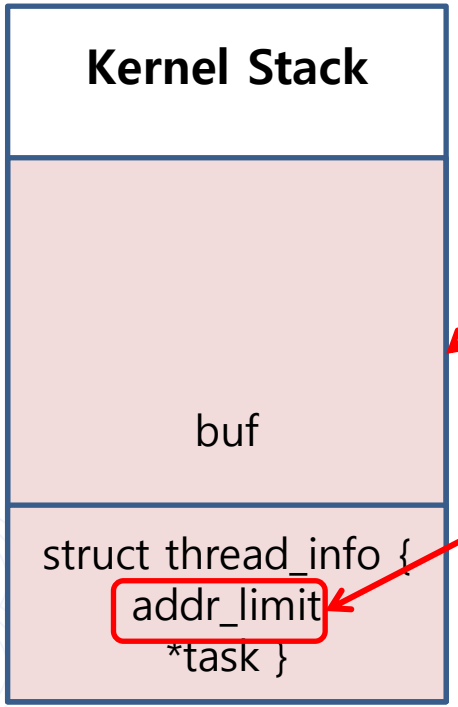
Vulnerability

```
int vul_func(int size, int off, int val)
{
    int arr[size];
    ....
    for (i=0; i<size; i++)
        arr[i] = val;
    ....
}
```

Attack succeed? Depends on vulnerability.
Is it in real world?
→ CVE-2010-3848, CVE-2010-3850

Modify addr_limit via stack-based attack

Stack overflow – Type 4: Recursion



Vulnerability

```
int vul_func(char *str)
{
    char buf[64];
    ....
    if (~~)
        vul_func(str);
    ....
    strcpy(buf, str);
}
```

Attack succeed? Too difficult..
Is it in real world?
→ CVE-2016-1583

Modify addr_limit via stack-based attack

Stack overflow – Summary

- Type1 : Classic buffer overflow, Simple, No vulnerability these days
- Type2 : Out-of-bound index, Simple, No vulnerability these days

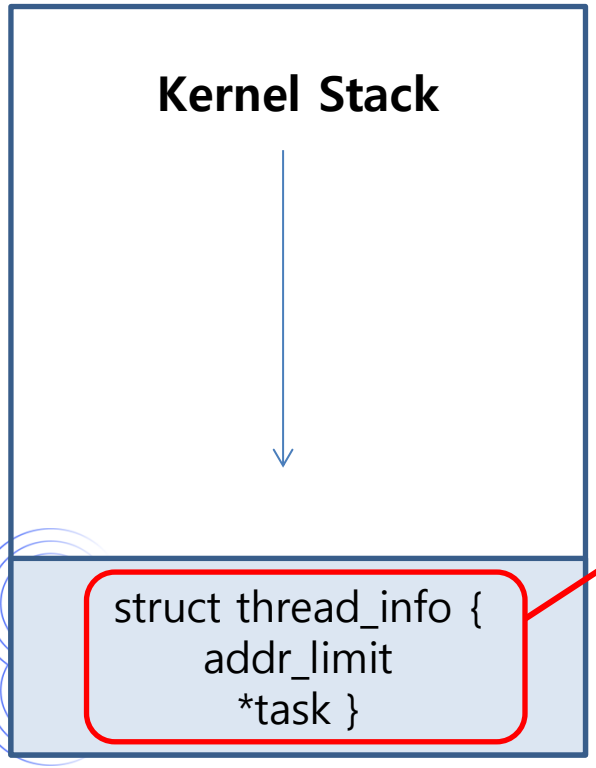
- Type3 : VLA (Variable Length Array), Complex, Real-world vulnerability
- Type4 : Recursion, Complex, Real-world vulnerability

➔ No more simple vulnerability!! Only remains complex vulnerability!!

Defense-4 :
Split addr_limit from stack

Split addr_limit from stack

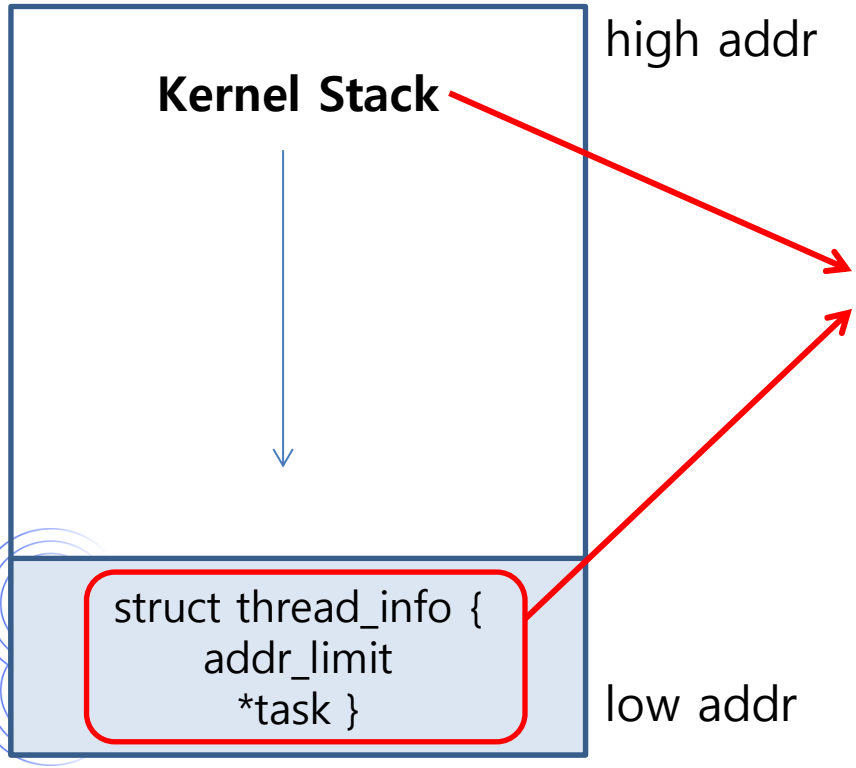
Why "struct thread_info" be in kernel stack??



If "struct thread_info" can be stored
somewhere not related to kernel stack,,
Safe against the previous stack-based attack!!

Split addr_limit from stack

Why "struct thread_info" be in kernel stack??



There is no meaningful relationship between Kernel stack and thread_info!
We can split them!!
It's a problem of SW design.

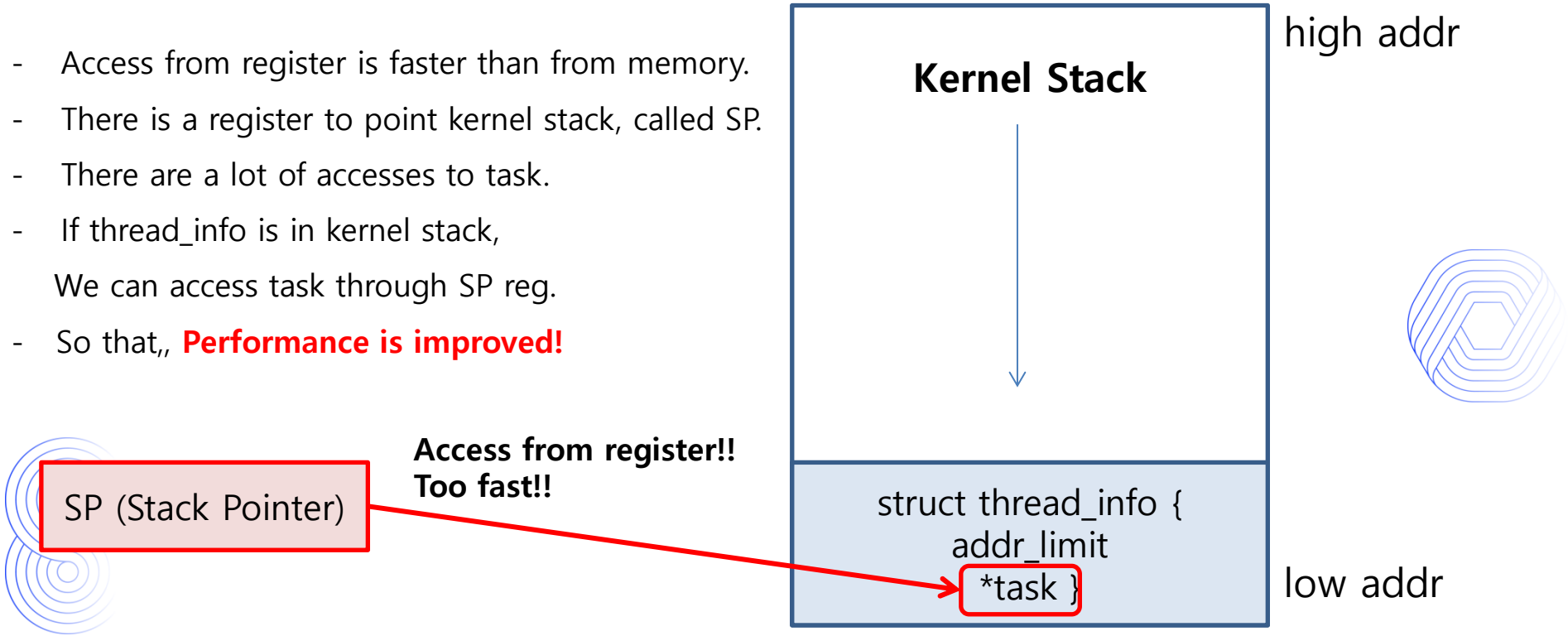
Wait.. Then,,

Why "struct thread_info" be in kernel stack?

Split addr_limit from stack

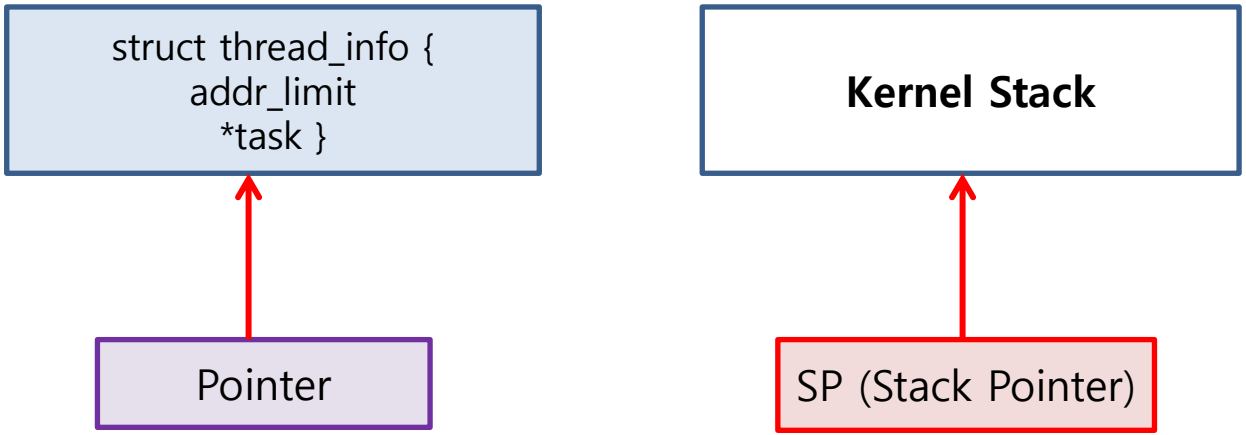
Stack pointer to point task_struct

- Access from register is faster than from memory.
- There is a register to point kernel stack, called SP.
- There are a lot of accesses to task.
- If thread_info is in kernel stack,
  We can access task through SP reg.
- So that,, **Performance is improved!**



Split addr_limit from stack

Split addr_limit from stack



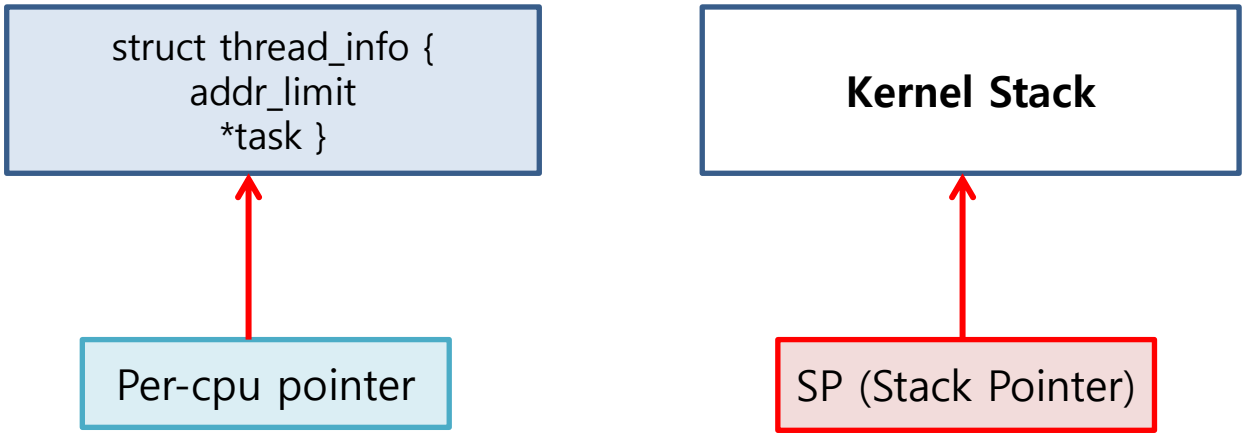
- "Pointer" is needed for pointing thread_info instead of SP.

Performance : Register → Memory

Security is ok, But..
Performance overhead here!!

Split addr_limit from stack

Optimization on Intel x86_64

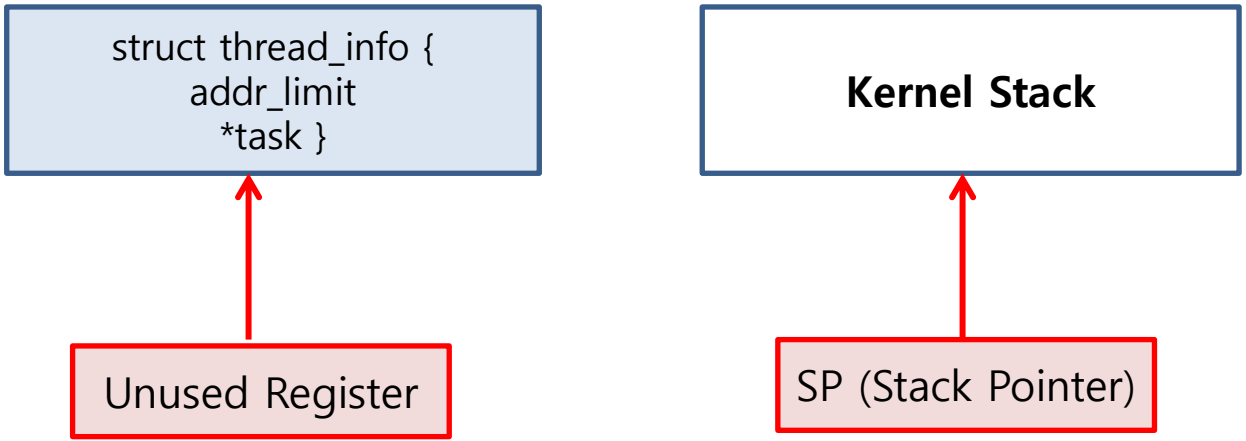


Performance : Register > Per-cpu > Memory

Security is ok, and Overhead is not bad!!

Split addr_limit from stack

Optimization on ARM 64

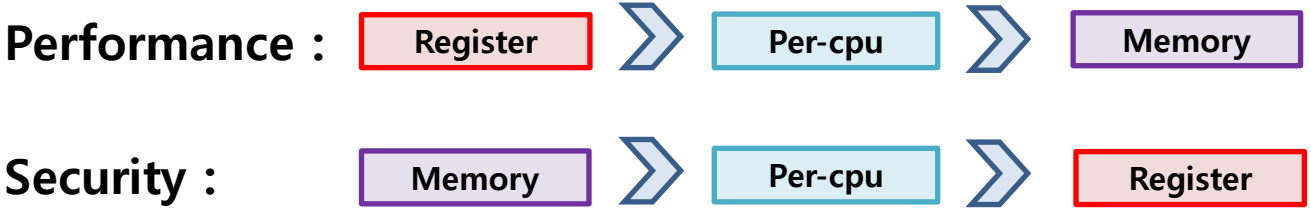


Performance : Register > Per-cpu > Memory

Security is ok, and Overhead is near zero!!

Split addr_limit from stack

Tradeoff between Security and Performance



Always there is a tradeoff between Performance and Security..

Are “Register” and “Per-cpu” really safe?? Hmm...

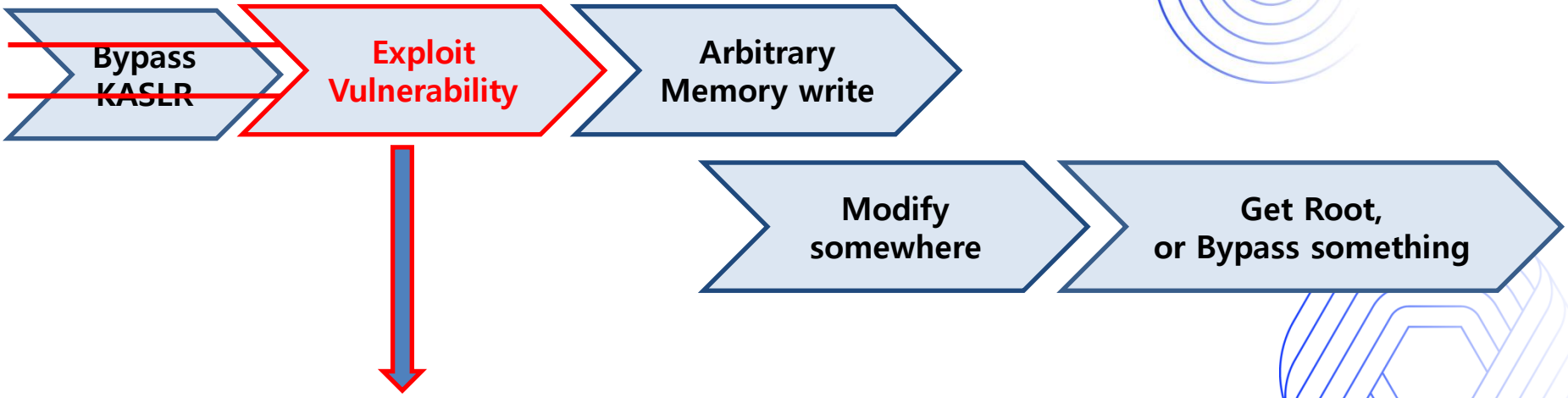
Split addr_limit from stack

Summary

Defense solution for fixing SW design problem
have to satisfy both
Security and Performance!!

+ Bonus :
Advanced attacks

Advanced attacks



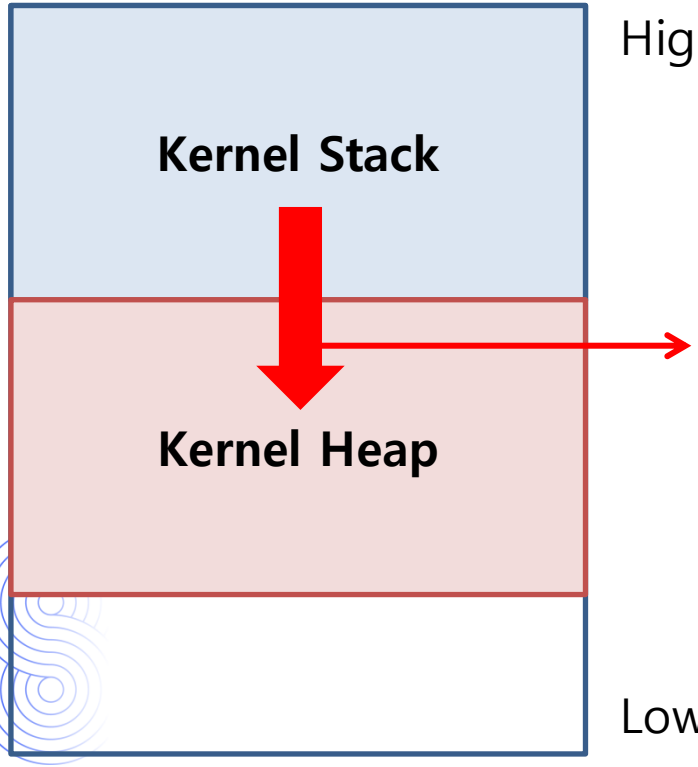
Advanced attacks

Pick two keywords of advanced attacks

Adjacent / Spraying

Advanced attacks

Adjacent, Type1 : Heap / Stack

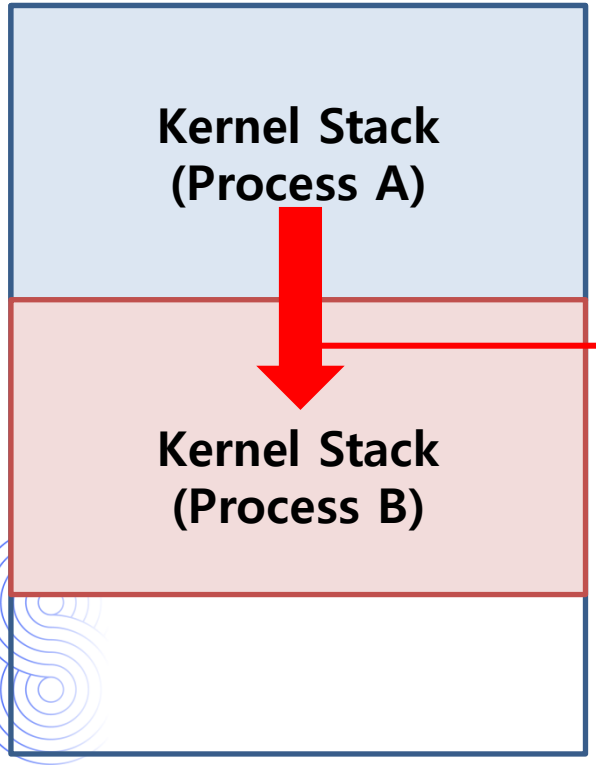


How about trigger overflows from Stack to Heap?
If stack is completely safe, We can consider this approach!

a.k.a. Large Memory Vulnerabilities, or Stack Clash

Advanced attacks

Adjacent, Type2 : Stack of Process A / Stack of Process B



High addr

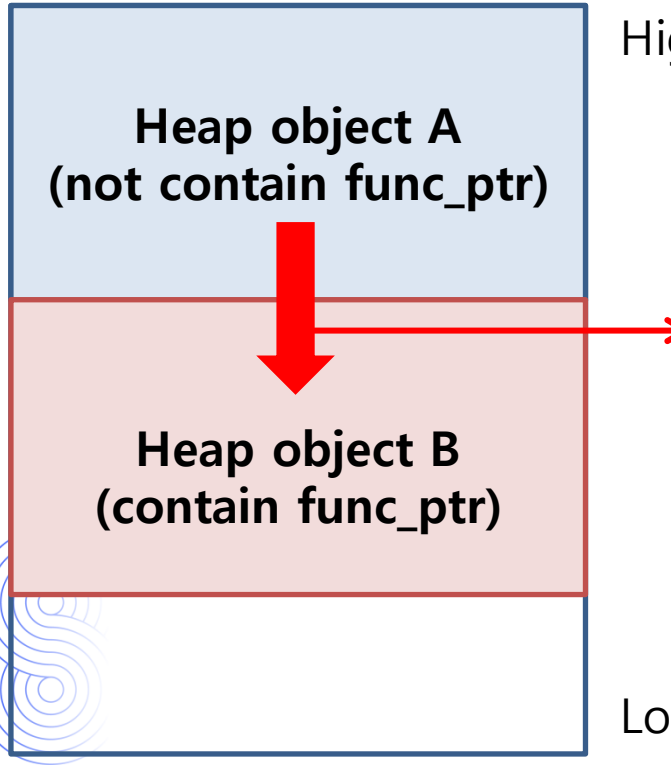
How about trigger overflows
from Process A's stack to Process B's stack?

- CVE-2010-3848, CVE-2010-3850

Low addr

Advanced attacks

Adjacent, Type3 : Heap object A / Heap object B



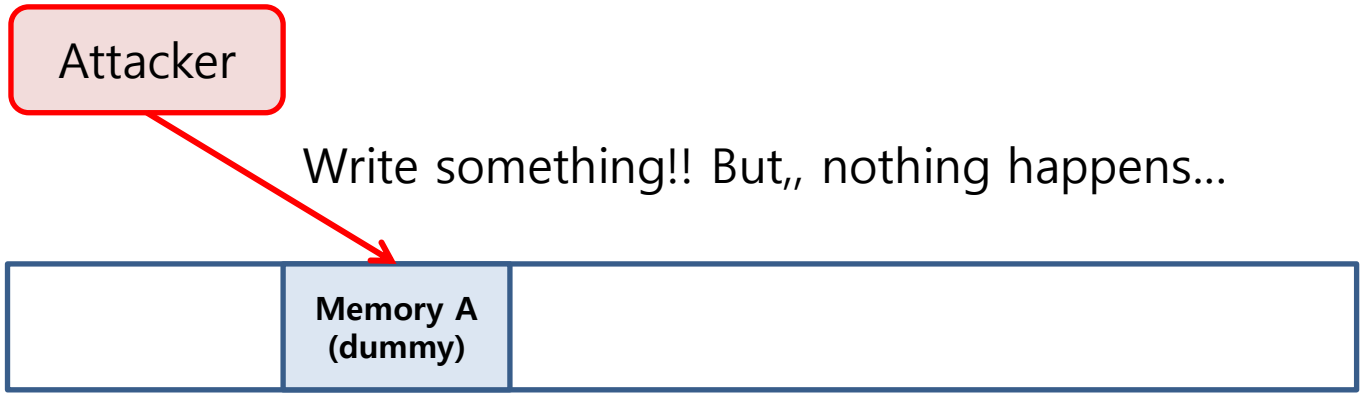
How about trigger overflows
from Heap object A to Heap object B?

Attacker can't modify func_ptr in object A,
But, Can modify func_ptr in object B!!

Advanced attacks

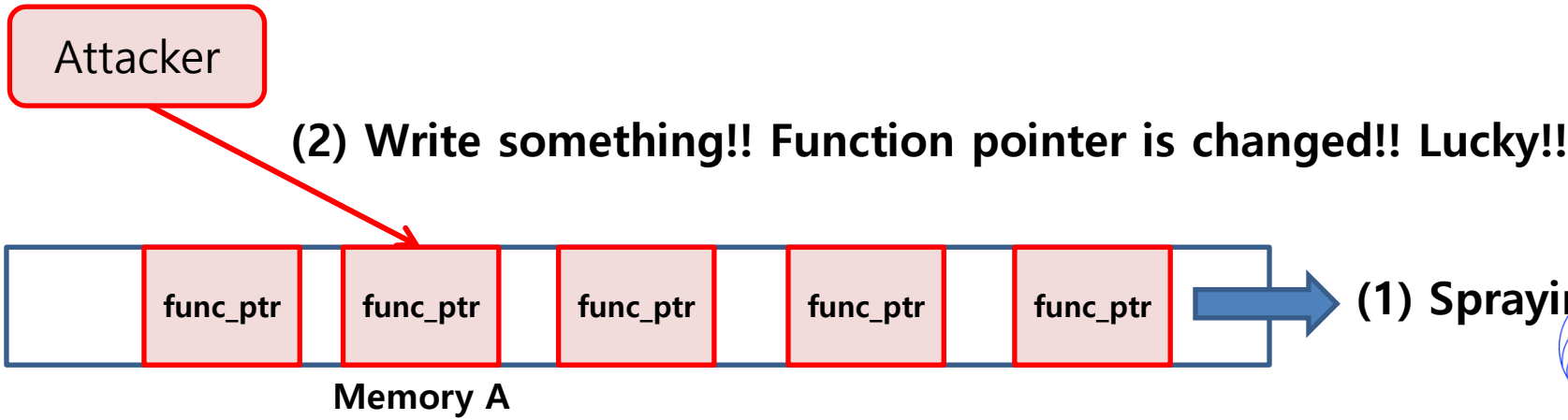
Spraying

- Assume that attacker get an ability to write value to kernel memory A.
- Kernel memory A is random address. Attacker doesn't know what here it is.



Advanced attacks

Spraying



THANK YOU

Sample exploit code is at

https://github.com/jinb-park/linux-exploit/tree/master/samples/adjacent-kstacks