SOSCON
Attack and Defense on Linux kernel

SAMSUNG Research | Security Team | Jinbum Park
2018.10.18
Full steps of attack on Linux kernel

**Attack-1**: Modify sensitive RW data
**Defense-1**: ro-after-init

**Attack-2**: Modify process credential
**Defense-2**: PrivWatcher

**Attack-3**: addr_limit bug
**Defense-3**: Add checking for addr_limit

**Attack-4**: Modify addr_limit via stack-based attack
**Defense-4**: Split addr_limit from stack

+ **Bonus**: Advanced attacks
Full steps of attack on Linux kernel
Full steps of attack on Linux kernel

1. Bypass KASLR
2. Exploit Vulnerability
3. Arbitrary Memory write
4. Modify somewhere
5. Get Root, or Bypass something
Exploit Linux kernel vulnerabilities.
By exploiting them, Attacker can
- Modify control flow,
- Do arbitrary memory write.
Full steps of attack on Linux kernel

- Bypass KASLR
- Exploit Vulnerability
- Arbitrary Memory write
- Modify somewhere
- Get Root, or Bypass something

- 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

- **Bypass KASLR**
- **Exploit Vulnerability**
- **Arbitrary Memory write**

**Modify somewhere**

- Get Root, or Bypass something

- 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

- Bypass KASLR
- Exploit Vulnerability
- Arbitrary Memory write
- Modify somewhere
- Get Root, or Bypass something

Final Goal!!
Attack-1: Modify sensitive RW data
Modify sensitive RW data

Bypass KASLR → Exploit Vulnerability → Arbitrary Memory write

Modify somewhere

Get Root, or Bypass something
Modify sensitive RW data

Sensitive RW data: Function Pointer

Why function pointer is critical?
Let’s look at Linux kernel 3.10.

```c
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

- 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.

```c
static struct sidtab sidtab;
struct policydb_policydb;
int ss_initialized;
```

Flag to represent whether SELinux is initialized or not.

```c
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

```c
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?

- 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

```c
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

```c
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
Reduce attack surface as much as possible!!
Attack-2: Modify process credential
Modify process credential

Bypass KASLR → Exploit Vulnerability → Arbitrary Memory write

Modify somewhere

Get Root, or Bypass something

Process credential
**Modify process credential**

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

    /* process credentials */
    const struct cred__rcu *real_cred; /* obj
    * credentials (COW) */
    const struct cred__rcu *cred;     /* effeci
    * 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 susid; /* saved UID of the task */
};
```

- 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)

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

Apply the new cred to current process

Make a new cred for root (uid=0)

- 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; /* effective
    * credentials (COW) */

  /* 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

Type 3: Modify cred itself

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

    /* process Credentials */
    const struct cred__rcu *real_cred; /* obj
    * _credentials (COW) */
    const struct cred__rcu *cred; /* effective
    * _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: 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

If It’s valid

If It’s not valid, drop the access.

- 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.
Is this merged in Linux kernel for Galaxy?

- Not same solution to PrivWatcher.
  But, There is a similar solution in after Galaxy S7.
You can add your security solution into your product!
Attack-3: addr_limit bug
addr_limit bug

Bypass KASLR → Exploit Vulnerability → Arbitrary Memory write → Modify somewhere → Get Root, or Bypass something

addr_limit bug → Read/Write all kernel memory
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.

```c
/*
 * 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`

- **User**: `addr_limit == USER_DS`
  - Can access user space only
  - Updated by Kernel or Kernel driver

- **Kernel**: `addr_limit == KERNEL_DS`
  - Can access user+kernel space
  - Restored by Kernel or Kernel driver

- **User**: `addr_limit == USER_DS`
  - Can access user space only
  - Updated by Kernel or Kernel driver
**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

```c
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;
    }
}
```

- 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: 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!!

User: `addr_limit == USER_DS`

Kernel: `addr_limit == KERNEL_DS`

Human error point!!

User: `addr_limit == KERNEL_DS`
Add checking for `addr_limit`

Solution

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

User: `addr_limit == USER_DS`

Kernel: `addr_limit == KERNEL_DS`

Checking!! Reporting error!!

Process will be killed!!

User: `addr_limit == KERNEL_DS`
Add checking for addr_limit

Solution

```c
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

Enforce security checking to eliminate human errors!!
Attack-4: Modify addr_limit via stack-based attack
Modify addr_limit via stack-based attack

- Bypass KASLR
- Exploit Vulnerability
- Arbitrary Memory write
- Modify somewhere
- Get Root, or Bypass something

Stack overflow → Modify addr_limit
Read/Write all kernel memory
Modify addr_limit via stack-based attack

Where “addr_limit” be stored? In kernel stack!!

Process descriptor

struct task_struct {
    *stack
}

Kernel stack per process

struct thread_info {
    addr_limit
    *task
}

Kernel Stack

high addr

low addr
How about trying stack overflow attack as a classic?

Kernel stack per process

```
struct thread_info {
    addr_limit *task
}
```

Normal writes

Overflows!!
Modify `addr_limit` via stack-based attack

Stack overflow – Type 1: classic buffer overflow

```c
struct thread_info {
    addr_limit
        *task }

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

Vulnerability

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

```c
struct thread_info {
    addr_limit *task
}

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

Vulnerability

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)

```
struct thread_info {
    addr_limit
        *task
}

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

Kernel Stack

- `buf` (variable)
- `struct thread_info { addr_limit *task }` (structure)

Vulnerability

```c
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??

Kernel Stack

```
struct thread_info {
    addr_limit *task
}
```

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??

Kernel Stack

struct thread_info {
  addr_limit
  *task
}

high addr

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 \texttt{addr\_limit} from stack

Stack pointer to point \texttt{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 \texttt{thread\_info} is in kernel stack, we can access task through SP reg.
- So that, \textit{Performance is improved!}

\begin{itemize}
  \item Access from register!!
  \item Too fast!!
\end{itemize}
Split addr_limit from stack

- “Pointer” is needed for pointing thread_info instead of SP.

Performance:

Security is ok, But...

Performance overhead here!!
Split addr_limit from stack
Optimization on Intel x86_64

```c
struct thread_info {
    addr_limit *task;
};
```

Kernel Stack

Per-cpu pointer

SP (Stack Pointer)

Performance:
- Register
- Per-cpu
- Memory

Security is ok, and Overhead is not bad!!
Split addr_limit from stack

Optimization on ARM 64

```
struct thread_info {
    addr_limit *task
}
```

Kernel Stack

Unused Register

SP (Stack Pointer)

Performance:

- Register
- Per-cpu
- Memory

Security is ok, and Overhead is near zero!!
Always there is a tradeoff between Performance and Security..

Are “Register” and “Per-cpu” really safe?? Hmm...
Defense solution for fixing SW design problem have to satisfy both Security and Performance!!
+ Bonus: Advanced attacks
Advanced attacks

Bypass KASLR → Exploit Vulnerability → Arbitrary Memory write → Modify somewhere → Get Root, or Bypass something

Focusing on here!
Advanced attacks

Pick two keywords of advanced attacks

Adjacent  /  Spraying
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

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

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

Write something!! But,, nothing happens...
Advanced attacks

Spraying

(1) Spraying!!

Attacker

(2) Write something!! Function pointer is changed!! Lucky!!

Memory A
THANK YOU

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