Proposal of a Method to Prevent Privilege Escalation Attacks for Linux Kernel

2017/9/15

Yuichi Nakamura, Ph.D., Hitachi, Ltd.
Toshihiro Yamauchi, Ph.D., Okayama University
Contents

1. Introduction
2. Design and implementation
3. Evaluation and demo
4. Remaining issues and future direction
1. Introduction
Privilege escalation

- In traditional Linux, root(uid=0) can do everything

- Attackers seeks to get the root shell exploiting “privilege escalation vulnerabilities”.

- Especially, Linux kernel vulnerabilities are often exploited.
  - Only 2017/1/1-8/1, 5 exploit codes for privilege escalation are disclosed in exploitdb.com

- MAC (Mandatory Access Control) technologies had been introduced into Linux to confine root.
  - SELinux, AppArmor, Smack...
Typical process of privilege escalation exploiting kernel

User space

(1) Exec Exploit program

Kernel space

(2) Call syscall

(3) Exploit Vulnerabilities and $uid=0$

(4) Exit syscall

User space

(5) Launch shell ($uid=0$)

Usually difficult to write long exploit codes, So just changes $uid=0$
commit_creds(prepare_kernel_creds(0));

Does anything using root shell

MAC confines this
MAC is great, but not enough for kernel exploits

- MAC is often disabled
  - Unfortunately, it is a fact 😞
  - Due to difficulties to create, manage security policies.

- MAC can be bypassed when Linux kernel vulnerability is exploited
  - E.g.: bypassing SELinux
    Just overwrite the address of “selinux_enforcing” as “0”

- MAC policy is not configured for login users by default.
  - SELinux
    “unconfined_t” (allowed almost everything) is assigned to login users
  - AppArmor
    Login users are not confined by default

Motivation of our work:
Prevent privilege escalation via Linux kernel vulnerabilities even without MAC
2. Design and implementation
Design goal

1) Prevent privilege escalation exploiting vulnerabilities in the Linux kernel
   • Not 100% protection, but reduce chance, make exploit difficult

2) Small performance impact

3) No impact to system administration
   • Zero configuration

4) Simple implementation
   • Avoid modification to existing data structure, functions
Basic concept

- Very few system calls change credential information (setuid, setgid..)
- Other system calls should not change credentials.

The concept is implemented for x86_64 arch
Proposed method: AKO (Additional Kernel Observer)

1. **Process**
   - System call service routine
   - **System call**
     - **Yes** (e.g., setuid)
     - **NO**
   - **End of System call**

2. **AKO extension**
   - Is the syscall Change credential normally?
     - **No**
       - Store current credential
       - Check the change of credential
         - Is credential Changed?
           - **No**
             - **Audit and exit**
           - **Yes**
             - **Credential information**
               - Zero configuration
               - Does not change existing interface

**User space**

**Kernel space**
Implementation: Entry of syscall

Process

System call service routine

END OF SYSTEM CALL

USER SPACE

KERNEL SPACE

AKO extension

Is the syscall Change credential normally?

NO

Store current credential

Check the change of credential

Is credential Changed?

NO

YES

Audit and exit

Credential information
Hook syscalls

- To hook all syscalls, entry of syscalls has to be modified. Hook function AKO_before is called.
- In the hook functions, syscalls that may change credential (uid, gid, capabilities) are not checked.

* arch/x86/entry/entry_64.S

```assembly
ENTRY(entry_SYSCALL_64)
...
call AKO_before
...
call *sys_call_table(, %rax, 8)
...
```

* arch/x86/kernel/ako.c

```c
asmlinkage void AKO_before
(struct ako_struct * ako_cred, unsigned long long ako_sysnum) {
...
if((sysnum == __NR_execve) || (sysnum == __NR_setuid) || (sysnum ==
   __NR_setgid) || (sysnum == __NR_setreuid) ||
   (sysnum == __NR_setregid) || (sysnum == __NR_setresuid) || (sysnum ==
   __NR_setresgid) || (sysnum == __NR_setfsgid) || (sysnum == __NR_setfsuid) ||
   (sysnum == __NR_setfsgid) || (sysnum == __NR_capset) || (sysnum ==
   __NR_prctl) || (sysnum == __NR_unshare) ){
    return 0;
...
```
A struct `ako_struct` is prepared to store credential information.

```c
#include/linux/ako.h

struct ako_struct {
    unsigned long ako_addr_limit;
    uid_t ako_uid;
    uid_t ako_euid;
    uid_t ako_fsuid;
    uid_t ako_suid;
    gid_t ako_gid;
    gid_t ako_egid;
    gid_t ako_fsgid;
    gid_t ako_sgid;
    __u32 ako_inheritable[2];
    __u32 ako_permitted[2];
    __u32 ako_effective[2];
    __u32 ako_bset[2];
};
```

- **UID, GID**: Trivial
- **Capabilities**: DAC_OVERRIDE can avoid permission check
- **addr_limit**: This is used for privilege escalation by changing limit between user/kernel space address.
Saving credentials: embedded cred into stack

- A struct `ako_struct` is prepared to store credential information
- `ako_struct` is embedded in unused area of kernel stack for syscall

* `arch/x86/entry/entry_64.S`

```assembly
ENTRY(entry_SYSCALL_64)
...
<ako_struct is embedded here>
call AKO_before
...
call *sys_call_table(, %rax, 8)
...
```

* `arch/x86/kernel/ako.c`

```c
asmlinkage void AKO_before
(struct ako_struct * ako_cred, unsigned long long ako_sysnum){
...
ako_cred->ako_uid = current->cred->uid.val;
ako_cred->ako_euid = current->cred->euid.val;
ako_cred->ako_fsuid = current->cred->fsuid.val;
...}
```

- `%rdi` (first arg for `ako_before`) is set here
- `cred info before syscall is called is saved`
Implementation: exit of syscall

Process

System call service routine

AKO extension

Is the syscall Change credential normally?

Store current credential

Check the change of credential

Is credential Changed?

Audit and exit

User space

Kernel space

End of System call

© Hitachi, Ltd. 2017. All rights reserved.
Check change of credential information

* arch/x86/entry/entry_64.S

ENTRY(entry_SYSCALL_64)
...
call AKO_before
...
call *sys_call_table(,%rax,8)
...
<set %rdi to addr of ako_struct>
call AKO_after

* arch/x86/kernel/ako.c

asmlinkage void AKO_after(struct ako_struct * ako_cred)
{
if(ako_cred->ako_uid != current->cred->uid.val ||
ako_cred->ako_euid != current->cred->euid.val ||
ako_cred->ako_fsuid != current->cred->fsuid.val ||
ako_cred->ako_suid != current->cred->suid.val){
    audit_AKO_uid(ako_cred);
    uid_modified = 1;
}
...
if (uid_modified) {
    do_exit(SIGKILL);
...
uid is changed, it is attack attempt so, exit forcefully
Expansion to watch disabling MAC

Attack attempt to disable MAC can also be watched.

Example: SELinux
- Watch the change of sid, exec_sid, selinux_enforcing
3. Evaluation, Demo
Evaluation against design goal

**Design Goal**

1) Prevent privilege escalation exploiting vulnerabilities in the Linux kernel
   - Not 100% protection, but reduce chance, make exploit difficult

2) Small performance impact

3) No impact to system administration
   - Zero configuration → Achieved

4) Simple implementation
   - Avoid modification to existing data structure, functions
     → Achieved

Experiment needed
Experiments

1. Preventing attack
   * See whether AKO can prevent privilege escalation attacks using PoC exploit codes

2. Performance test
   * Measure the overhead on system call
## Experiment result #1: Preventing attacks

- **Tried 6 PoC codes, 5/6 are prevented**

<table>
<thead>
<tr>
<th>#</th>
<th>CVE</th>
<th>Overview of vulnerability</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CVE-2013-1763</td>
<td>Array index error due to inadequate parameter check in socket()</td>
<td>Prevented at sendto syscall</td>
</tr>
<tr>
<td>2</td>
<td>CVE-2014-0038</td>
<td>Memory destruction due to inadequate parameter check in recvmsg()</td>
<td>Prevented at open syscall</td>
</tr>
<tr>
<td>3</td>
<td>CVE-2014-3153</td>
<td>Inadequate address check for re-queuing operation in futex()</td>
<td>Prevented at futex syscall</td>
</tr>
<tr>
<td>4</td>
<td>CVE-2016-0728</td>
<td>Use of integer overflow and freed memory in keyctl()</td>
<td>Prevented at keyctl syscall</td>
</tr>
<tr>
<td>5</td>
<td>CVE-2016-5195</td>
<td>a race condition occurs during a copy-on-write process(Dirtycow)</td>
<td>NG</td>
</tr>
<tr>
<td>6</td>
<td>CVE-2017-6074</td>
<td>Mishandles DCCP PKT REQUEST packet data in dccp rcv state process()</td>
<td>Prevented at recvfrom syscall</td>
</tr>
</tbody>
</table>

- **DirtyCow can not be prevented, because exploit code can do harm even without setting uid=0**
Experiment #2: Performance test

• Compared performance before and after introducing AKO.

• Environment
  • CPU: Intel Core i5-3470 3.2 GHz (4 cores)
  • Memory: 4.0 GB
  • OS: Linux 3.10.0 (64 bit)

• Microbench
  • Processing time of system calls

• Apache bench, kernel build time
Experiment #2: Performance test result (microbench)

LMBench:

<table>
<thead>
<tr>
<th>System call</th>
<th>Before (us)</th>
<th>After (us)</th>
<th>Overhead (us)</th>
</tr>
</thead>
<tbody>
<tr>
<td>stat</td>
<td>0.368</td>
<td>0.383</td>
<td>0.015</td>
</tr>
<tr>
<td>fstat</td>
<td>0.099</td>
<td>0.111</td>
<td>0.012</td>
</tr>
<tr>
<td>write</td>
<td>0.105</td>
<td>0.141</td>
<td>0.036</td>
</tr>
<tr>
<td>read</td>
<td>0.078</td>
<td>0.110</td>
<td>0.032</td>
</tr>
<tr>
<td>getppid</td>
<td>0.040</td>
<td>0.048</td>
<td>0.008</td>
</tr>
<tr>
<td>open+close</td>
<td>1.130</td>
<td>1.190</td>
<td>0.030</td>
</tr>
</tbody>
</table>
Experiment #2: performance test result

Apache bench: Processing time per request

<table>
<thead>
<tr>
<th>File size (KB)</th>
<th>Before (ms)</th>
<th>After (ms)</th>
<th>Overhead (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.465</td>
<td>0.467</td>
<td>0.002 (+0.4%)</td>
</tr>
<tr>
<td>10</td>
<td>0.638</td>
<td>0.640</td>
<td>0.002 (+0.3%)</td>
</tr>
<tr>
<td>100</td>
<td>1.523</td>
<td>1.525</td>
<td>0.002 (+0.1%)</td>
</tr>
</tbody>
</table>

Kernel build time

<table>
<thead>
<tr>
<th>Before (s)</th>
<th>After (s)</th>
<th>Overhead (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2669.0</td>
<td>2675.0</td>
<td>6.0(+0.2%)</td>
</tr>
</tbody>
</table>
4. Remaining issues and future direction
Security Consideration

- This mechanism can prevent existing exploit code. i.e. `commit_creds(prepare_kernel_cred(0));`

- However, after the mechanism is known to attackers, they will try to bypass it.

- Current implementation is not strong yet. -> Working now
Bypassing current implementation

Attackers can bypass the mechanism if ako_struct is overwritten in the exploit codes.

-> ako_struct should be stored more strongly.
Idea: randomizing address

At the start of entry_SYSCALL_64, insert random size padding

```
struct pt_regs

padding

ako_struct

Thread_info

kernel stack for syscall
```

Current status:
Begun prototype implementation.
Seems that some parts of kernel codes assumes that syscall kernel stack begins with struct pt_regs, and should be modified.
Conclusions

* Implemented a prototype to prevent privilege escalation attack

* Evaluated the performance impact, and effectiveness against existing attacks

* Remaining work
  - Tough implementation not to be bypassed
Trademarks

- Linux is the registered trademark of Linux Torvalds in the U.S. and other countries.

- Apache is the registered trademark of the Apache Software Foundation in the U.S. and other countries.

- Other brand names and product names used in this material are trademarks, registered trademarks, or trade names of their respective holders.
HITACHI
Inspire the Next