Trace Me If You can:
Bypassing Linux Syscall Tracing

Dr. Rex Guo, Lacework
Dr. Junyuan Zeng, Linkedin.com
About Rex Guo

• Principal Engineer @ Lacework
  • Behavior anomaly detection (Polygraph)
  • CSPM

• Engineering Manager @ Startups
  • Confluera (XDR)
  • Tetration (CWPP, now part of Cisco)

• Conference speaker at Blackhat, DEFCON,...

• @Xiaofei_REX
About Junyuan Zeng

• Linkedin
  • Senior Software Engineer: Kubernetes

• JD.com
  • Staff Security Architect/Engineer: Cloud native security

• Samsung Research America & FireEye
  • Staff Security Software Engineer/Researcher: Mobile security
An Incident - An Attacker’s View

Log4shell RCE on joe-box and executed a reverse shell
An Incident - An Attacker’s View

Log4shell RCE on joe-box and executed a reverse shell

Privilege escalation using sudo CVE-2021-3156
An Incident - An Attacker’s View

Log4shell RCE on joe-box and executed a reverse shell

read /etc/shadow

Privilege escalation using sudo CVE-2021-3156
An Incident - An Attacker’s View

Log4shell RCE on joe-box and executed a reverse shell

read /etc/shadow

Privilege escalation using sudo CVE-2021-3156

Read ssh process environment variable
An Incident - An Attacker’s View

Log4shell RCE on joe-box and executed a reverse shell

Privilege escalation using sudo CVE-2021-3156

read /etc/shadow

Read ssh process environment variable

Lateral movement to alice-box with ssh hijacking
An Incident - An Attacker’s View

Log4shell RCE on joe-box and executed a reverse shell

read /etc/shadow

Privilege escalation using sudo CVE-2021-3156

Read ssh process environment variable

Lateral movement to alice-box with ssh hijacking

?
An Incident - A Defender’s View

Log4shell RCE on joe-box and executed a reverse shell

Privilege escalation using sudo CVE-2021-3156

Read /etc/shadow

Lateral movement to alice-box with ssh hijacking

connect etc.
execve(at)
open(at)
open(at)
connect, etc.
Incident Response
Detection Rule Example

rule: untrusted program reads /etc/shadow

condition:

syscall == open(at)
and has read permission
and filename == /etc/shadow
and program is not in allowlist
Agenda

- Syscall Tracing
- Vulnerabilities
- Exploitations
- Mitigations
- Takeaways
System Call Tracing
System Call Tracing – Tracing Program

- Tracing programs collect system call data, e.g., arguments
- Tracing programs can “attach” to different hooks
  - tracepoints, kprobe, ptrace etc.
- Tracing programs implementations
  - Linux native mechanisms: ftrace, perf_events etc.
  - Kernel modules, eBPF probe and user space programs
System Call Tracing – tracepoint

- tracepoint
  - Kernel static hook
  - Linux kernel provides `sys_enter` and `sys_exit`

- Low overhead but only static interceptions
System Call Tracing – kprobe

• kprobe
  • Dynamic hook in the kernel
  • Register tracing programs on instructions in syscall code path
  • Dynamic but slow compared to tracepoint and need to know exactly how data is placed on the stack and registers
System Call Tracing – ptrace

- ptrace
  - A static hook
  - No Kernel Module/eBPF program are needed
  - Performance overhead is high
  - Can combine with seccomp to reduce overhead
- Others (LD_PRELOAD etc.)
Cloud Workloads

- Virtual machines
  - AWS EC2 instances
  - Google VM instances
- Containers on customer-managed VMs
  - AWS EC2 tasks
  - Standard GKE workloads (e.g. DaemonSet etc.)
  - AKS workloads
- Serverless containers: have no access to the host
  - AWS Fargate tasks
  - GCP Cloud Run services
- Others (AWS Lambda etc.)
## System Call Tracing for Cloud Workloads

<table>
<thead>
<tr>
<th>Workload</th>
<th>System Call Tracing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>VMs</strong></td>
<td>• Hooks: tracepoint, kprobe, ptrace</td>
</tr>
<tr>
<td></td>
<td>• Tracing programs: kernel programs (eBPF, kernel Module), user programs</td>
</tr>
<tr>
<td></td>
<td>• Tools: Falco eBPF/kernel Module, Falco pdig</td>
</tr>
<tr>
<td><strong>Containers</strong></td>
<td>• Hooks: tracepoint, kprobe, ptrace</td>
</tr>
<tr>
<td></td>
<td>• Tracing programs: kernel programs (eBPF, kernel Module), user programs</td>
</tr>
<tr>
<td></td>
<td>• Tools: Falco eBPF/kernel Module, Falco pdig</td>
</tr>
<tr>
<td><strong>Serverless</strong></td>
<td>• Hooks: ptrace</td>
</tr>
<tr>
<td><strong>Containers</strong></td>
<td>• Tracing programs: user programs</td>
</tr>
<tr>
<td></td>
<td>• Tools: Falco pdig</td>
</tr>
</tbody>
</table>
Open Source Projects

• Falco
  • Open source endpoint security monitoring project in CNCF
  • 5K+ github stars
  • Falco supports syscall tracing techniques:
    ● tracepoint + kernel module
    ● tracepoint + eBPF probe
    ● pdig: ptrace + userspace program

• Falco pdig
  • Support syscall tracing of serverless workloads
TOCTOU in Syscall Tracing

- `sys_connect(int fd, struct sockaddr __user * uservaddr, int addrlen)`
- TOC (Time-Of-Check): tracing programs dereference this user space pointer
- TOU (Time-Of-Use): the kernel dereferences this user space pointer
TOCTOU - Connect

Syscall Table (x86_64)
...
42 sys_connect
43 sys_accept
44 sys_sendto
...

ptrace_report_syscall(regs, message)
secure_computing(struct seccomp_data{regs...})
trace_sys_enter(regs, regs->orig_ax)

ptrace/seccomp/sysenter tracepoint

long __sys_connect((int fd,
struct sockaddr __user *uservaddr, int addrlen))
{
...
struct filename *tmp;
ret = move_addr_to_kernel
(uservaddr, addrlen, &address);
if (!ret)
ret = __sys_connect_file
(f.file, &address, addrlen, 0);
...
}

trace_sys_exit(regs, regs->ax)
ptrace_report_syscall(regs, message)
TOCTOU - Connect

User Space Kernel 5.7.0

syscall enter

syscall exit

ptrace_report_syscall(regs, message)
__secure_computing(struct seccomp_data{regs...})
trace_sys_enter(regs, regs->orig_ax)

Syscall Table (x86_64)
...
42 sys_connect
43 sys_accept
44 sys_sendto
...

long __sys_connect((int fd,
               struct sockaddr __user *uservaddr, int addrlen))
{
  ... 
  struct filename *tmp;
  ret = move_addr_to_kernel
         (uservaddr, addrlen, &address);
  if (!ret)
    ret = __sys_connect_file
           (f.file, &address, addrlen, 0);
  ...
}

trace_sys_exit(regs, regs->ax)
ptrace_report_syscall(regs, message)
TOCTOU - Connect

```
ptrace_report_syscall(regs, message)
__secure_computing(struct seccomp_data{regs...})
trace_sys_enter(regs, regs->orig_ax)
```

**Syscall Table (x86_64)**

```
42 sys_connect
43 sys_accept
44 sys_sendto
```

```
long __sys_connect((int fd,
   struct sockaddr __user *uservaddr, int addrlen))
{
   ...
   struct filename *tmp;
   ret = move_addr_to_kernel
   (uservaddr, addrlen, &address);
   if (!ret)
      ret = __sys_connect_file
       (f.file, &address, addrlen, 0);
   ...
}
```

```
trace_sys_exit(regs, regs->ax)
ptrace_report_syscall(regs, message)
```
TOCTOU - Connect

syscall enter

ptrace_report_syscall(regs, message)
__secure_computing(struct seccomp_data{regs...})
trace_sys_enter(regs, regs->orig_ax)

Syscall Table (x86_64)
...
42 sys_connect
43 sys_accept
44 sys_sendto
...

42 long __sys_connect((int fd,
 struct sockaddr __user *uservaddr, int addrlen))
{
...
 struct filename *tmp;
 ret = move_addr_to_kernel
 (uservaddr, addrlen, &address);
 if (!ret)
 ret = __sys_connect_file
 (f.file, &address, addrlen, 0);

...

trace_sys_exit(regs, regs->ax)
ptrace_report_syscall(regs, message)

Userspace pointer pointing to “socket address”
TOCTOU - Connect

syscall enter

ptrace_report_syscall(regs, message)
__secure_computing(struct seccomp_data{regs...})
trace_sys_enter(regs, regs->orig_ax)

Syscall Table (x86_64)
...
42 sys_connect
43 sys_accept
44 sys_sendto
...

---

Userspace pointer pointing to “socket address”

Kernel pointer pointing to “socket address”

syscall exit

trace_sys_exit(regs, regs->ax)
ptrace_report_syscall(regs, message)
TOCTOU - Connect

ptrace_report_syscall(regs, message)
__secure_computing(struct seccomp_data{regs...})
trace_sys_enter(regs, regs->orig_ax)

Syscall Table (x86_64)
... 42 sys_connect
43 sys_accept
44 sys_sendto
...

trace_sys_exit(regs, regs->ax)
ptrace_report_syscall(regs, message)
TOCTOU - Connect

User Space Kernel 5.7.0

Syscall Table (x86_64)

- sys_connect
- sys_accept
- sys_sendto

Execution Flow

syscall enter

ptrace_report_syscall(regs, message)
__secure_computing(struct seccomp_data{regs...})
trace_sys_enter(regs, regs->orig_ax)

Syscall Table (x86_64)

42 sys_connect
43 sys_accept
44 sys_sendto

long __sys_connect((int fd,
                     struct sockaddr __user *uservaddr, int addrlen))
{
    ...
    struct filename *tmp;
    ret = move_addr_to_kernel
         (uservaddr, addrlen, &address);
    if (!ret)
        ret = __sys_connect_file
              (f.file, &address, addrlen, 0);
    ...
}

trace_sys_exit(regs, regs->ax)
ptrace_report_syscall(regs, message)
TOCTOU - Connect

```
ptrace_report_syscall(regs, message)
__secure_computing(struct seccomp_data{regs...})
trace_sys_enter(regs, regs->orig_ax)

sys_enter_tracepoint
ptrace

kprobe

Syscall Table (x86_64)
... 42 sys_connect
  43 sys_accept
  44 sys_sendto
...

long __sys_connect((int fd,
  struct sockaddr __user *user_addr, int addrlen))
{
  ...
  struct filename *tmp;
  ret = move_addr_to_kernel
    (user_addr, addrlen, &address);
  if (!ret)
    ret = __sys_connect_file
      (f.file, &address, addrlen, 0);
  ...
}

trace_sys_exit(regs, regs->ax)
ptrace_report_syscall(regs, message)
```
**TOCTOU - Connect**

```
ptrace_report_syscall(regs, message)
__secure_computing(struct seccomp_data{regs...})
trace_sys_enter(regs, regs->orig_ax)
```

**Syscall Table (x86_64)**

```
... 42 sys_connect
43 sys_accept  
44 sys_sendto ...
```

```c
long __sys_connect((int fd,  
    struct sockaddr __user *uservaddr, int addrlen))
{
...  
    struct filename *tmp;
    ret = move_addr_to_kernel  
    (uservaddr, addrlen, &address);
    if (!ret)  
        ret = __sys_connect_file  
        (f.file, &address, addrlen, 0);
...
}
```
TOCTOU - Connect

Syscall Table (x86_64)

... 42 sys_connect
43 sys_accept
44 sys_sendto
...

ptrace_report_syscall(regs, message)
_secure_computing(struct seccomp_data{regs...})
trace_sys_enter(regs, regs->orig_ax)

TOU by Linux Kernel

Falco pdig

Syscall Table (x86_64)

... 42 sys_connect
43 sys_accept
44 sys_sendto
...

long __sys_connect((int fd,
     struct sockaddr __user *uservaddr, int addrlen))
{
    ...
    struct filename *tmp;
    ret = move_addr_to_kernel
        (uservaddr, addrlen, &address);
    if (!ret)
        ret = __sys_connect_file
            (f.file, &address, addrlen, 0);
    ...
}

Falco (<0.31.1) kernel module/eBPF

Falco pdig

trace_sys_exit(regs, regs->ax)
ptrace_report_syscall(regs, message)
TOCTOU Windows across Kernels

- TOCTOU windows exist since the initial release of tracepoint/ptrace
- Expected behaviors
- Monitor kernel memory

**tracepoint and ptrace have TOCTOU issues!**

**We knew! They are designed for perf/debug**

*Security SW Developers*  *Kernel Developers*
TOCTOU – Falco

• User space pointers are dereferenced by
  • `sys_exit` tracepoint (kernel module, eBPF)
  • `sys_exit` ptrace (pdig)

• Falco older than v0.31.1
  • Check with vendors which commercial versions are affected

• 12/06/2021 Issue reported ([CVE-2022-26316](CVE-2022-26316))

• 03/11/2022 Mitigation implemented ([Advisory](Advisory))
  • For selected system calls, compare `sys_enter` and `sys_exit` tracepoint data (Falco LKM, eBPF)
  • Compare `sys_enter` and `sys_exit` ptrace data (Falco pdig)
**TOCTOU – Falco**

- We evaluated the important syscalls in [Falco rules](#).

<table>
<thead>
<tr>
<th>Syscall</th>
<th>Category</th>
<th>TOCTOU?</th>
<th>Exploitable by blocking condition</th>
<th>Exploitable by DC29 attack</th>
</tr>
</thead>
<tbody>
<tr>
<td>connect</td>
<td>Network</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>sendto/sendmsg</td>
<td>Network</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>open(at)</td>
<td>File</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>execve</td>
<td>File</td>
<td>N</td>
<td>N*</td>
<td>N*</td>
</tr>
<tr>
<td>rename</td>
<td>File</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>renameat(2)</td>
<td>File</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>mkdir(at)</td>
<td>File</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>rmdir</td>
<td>File</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>unlink(at)</td>
<td>File</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>symlink(at)</td>
<td>File</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>chmod/fchmod(at)</td>
<td>File</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>creat</td>
<td>File</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>
Exploit Requirements

- Exploitation requirements
  - No additional privilege and capabilities
  - Control the time to inject the delay
  - Enough delay for pointer overwrite
  - Reliable
Exploit Strategy 1 (DEFCON 29)

- Injected delay is small
- Requires Userfaultfd syscall for precise injection while pausing the kernel execution
- seccomp can block userfaultfd syscall (e.g., docker default seccomp profile)
- Falco’s mitigation was to detect userfaultfd

Exploit Strategy 2

- Injected delay >> the syscall execution time
- No Precise control is required
Syscall Built-in Delay

• Attackers can trigger significant syscall delays by introducing:
  • Blocking conditions (attack sys_exit)
  • Seccomp rules (attack sys_enter)

• Syscall can get “blocked”

<table>
<thead>
<tr>
<th>Categories</th>
<th>Syscalls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>fork/exec/exit/wait/…</td>
</tr>
<tr>
<td>File system</td>
<td>open(at)/symlink(at)/read/write/…</td>
</tr>
<tr>
<td>Networking</td>
<td>connect/accept/socket/…</td>
</tr>
<tr>
<td>Security</td>
<td>seccomp/keyctl/…</td>
</tr>
<tr>
<td>Many others…</td>
<td>…</td>
</tr>
</tbody>
</table>
Connect Syscall

Syscall Execution Flow:

- Client (monitored by tracing program)
- Server

- sys_enter connect
- SYN
- SYN, ACK
- ACK
- tracepoint/ptrace reads syscall argument
- sys_exit connect
Bypassing Connect Syscall Tracing (Demo)
Blocking Syscalls (File Systems)

- File system syscalls are all affected
  - open/openat
  - creat
  - rename/renameat/renameat2
  - mkdir/mkdirat
  - rmdir
  - Other file system syscalls with pointer arguments

- Other syscalls are also affected due to fetching files from file systems.
  - execve/execveat
Filesystem in USErspace - FUSE

• User space filesystem framework
• Used as remote storage FUSE
  • Access the remote files as local ones
  • Faster evolvement and don’t panic the kernel etc.
• Remote storage FUSE examples:
  • gcsfuse\textsuperscript{1}: developed by Google for GCS
  • s3fs-fuse\textsuperscript{2}: Amazon S3
  • BlobFuse\textsuperscript{3}: developed by Azure for Blob storage
  • MezzFS\textsuperscript{4}: developed and deployed @ Netflix
  • Many others (sshfs etc.)

---
\textsuperscript{1} https://github.com/GoogleCloudPlatform/gcsfuse
\textsuperscript{2} https://github.com/s3fs-fuse/s3fs-fuse
\textsuperscript{3} https://github.com/Azure/azure-storage-fuse
\textsuperscript{4} https://netflixtechblog.com/mezzfs-mounting-object-storage-in-netlicx-media-processing-platform-c0a0c446ba
Remote Storage FUSE - Architecture
Bypassing Openat Tracing (Demo)

Execution Flow

- **sys_entry openat**
  - *pathname -> page A (malicious file)*
  - requests are routed from kernel space to user space

- **pthread_create**
- **overwrite thread**
  - overwrite page A with benign file

- **sys_exit openat**

- **user space**
  - send request to remote file store

- **file store**
  - respond to the data access request

**client (monitored by tracing program)**

**S3, GCS, etc.**
TOCTOU – sys_enter (Connect)

ptrace_report_syscall(regs, message)
__secure_computing(struct seccomp_data{regs...})
trace_sys_enter(regs, regs->orig_ax)

ptrace

Syscall Table (x86_64)
...
42 sys_connect
43 sys_accept
44 sys_sendto
...

long __sys_connect((int fd,
    struct sockaddr __user *uservaddr, int addrlen))
{
    ...
    struct filename *tmp;
    ret = move_addr_to_kernel
        (uservaddr, addrlen, &address);
    if (!ret)
        ret = __sys_connect_file
            (f.file, &address, addrlen, 0);
    ...
}

trace_sys_exit(regs, regs->ax)
ptrace_report_syscall(regs, message)

TOU by Linux Kernel

ptrace

seccomp

Execution Flow
Seccomp Introduction

• Kernel level mechanism to restrict syscalls
• Modern sandboxes heavily relies on seccomp
• Developers can write rules to:
  • allow/block certain syscalls
  • allow/block syscalls based on argument values
• These rules can be quite complex (read more)
  • More rules takes more time to compute
• First inserted rules are evaluated last
Attacking Syscall Enter

Execution Flow

syscall thread

insert seccomp rules

sys_enter creat

*pathname -> page A (benign file)

ptrace reads syscall argument

seccomp rule check start

seccomp rule check ends

kernel reads the argument

sys_exit creat

pthread_create

overwrite thread

overwrite page A with malicious file

*pathname -> page A (malicious file)
ptrace + seccomp redirect

• Tracer starts App

New rules will be inserted at the beginning of the list

order of rule evaluation
## Exploitation and Mitigations

<table>
<thead>
<tr>
<th>Tracing location</th>
<th>TOCTOU Exploitation</th>
<th>Mitigations</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptrace @ sys_enter</td>
<td>Seccomp filter insertion</td>
<td>- ptrace + seccomp redirect to start the app.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Inspect seccomp filters already inserted when attaching to a running app</td>
</tr>
<tr>
<td>tracepoint @ sys_enter</td>
<td>Unreliable</td>
<td>N/A</td>
</tr>
<tr>
<td>tracepoint @ sys_exit</td>
<td>- Blocking syscall (This talk)</td>
<td>- Compare tracepoint sys_enter and sys_exit args</td>
</tr>
<tr>
<td></td>
<td>- Phantom attack v1 (DEFCON 29)</td>
<td></td>
</tr>
<tr>
<td>ptrace @ sys_exit</td>
<td>Same as above</td>
<td>- Deploy all mitigations for ptrace @ sys_enter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Compared the sys_enter and sys_exit syscall args</td>
</tr>
<tr>
<td>kprobe @ kernel internal</td>
<td>It depends</td>
<td>Read the kernel copy of the syscall args</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- LSM (BPF-LSM)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Other interfaces</td>
</tr>
</tbody>
</table>
Key Takeaways

1. Linux kernel tracing can be bypassed reliably
   • Check your security tools

2. Mitigation is complex (workload type and kernel compatibility)
   • Check your security tools’ mitigation claims

3. Correlate different data sources

4. Know your normal

• Discussing further?
  • @Xiaofei_REX / rex.guo *NOSPAM* lacework DOT com
  • jzeng04 *NOSPAM* gmail DOT com
  • POC: https://github.com/rexguowork/phantom-attack
Acknowledgement

• Joel Schopp (Linux kernel / Security)
• Lacework Labs
  • James Condon
  • Greg Foss
  • Chris Hall
  • Jared Stroud
• Falco open source team
  • Leonardo Di Donato
  • Michael Clark
  • Michele Zuccala
  • Luca Guerra
• John Dickson