Kernel security hacking for the Internet of Things

Daniel Sangorrin

IoT Technology center
Toshiba Corp. Industrial ICT solutions,
Kawasaki city, Kanagawa prefecture (Japan)
E-mail: daniel.sangorrin@toshiba.co.jp

LinuxCon Japan 2015
June 3rd, 2015
Tokyo
Kernel security hacking for the IoT

1. Introduction
2. Reducing the attack surface
3. Leveraging determinism
4. Protecting the critical software
5. Conclusions
About me

Real-time embedded systems engineer

- Started with real-time embedded software and drivers (8 years).
  - MaRTE OS (Ada95 RTOS), SafeG (ARM Trustzone monitor), TOPPERS/FMP (Japanese multi-core RTOS).
- Now, mostly customizing Linux for embedded devices (2 years).
  - Yocto-based project: META-DEBIAN (talk on Friday 5th, 16:20h)
  - Long-term Support Industrial (LTSI) kernels + Real-time patch

Not a security expert

- Trying to catch up with such a broad subject.

Hobbies

- Manga, Puramoderu, hiking, futsal, …
Purpose of this talk

Two main purposes

- Raise concern about the security of embedded systems in the IoT.
- Share a few things I learned while investigating Linux security and encourage you to try and share your own techniques.

*I won’t be talking about physical security today*
What’s (on with) the IoT?

IoT (my simplified definition)

➢ A distributed computing system consisting of:
  ▪ Embedded devices interacting with the physical world (Things) through sensors and actuators…
  ▪ and connected to the cloud (eg: smart servers, PCs, other devices) through a network (eg: a virtual private network)…
  ▪ in order to solve a problem or offer a service (eg: remote monitoring and control, optimization, automation, added value).

Interest over time (Japan)

Data Source: Google Trends (www.google.com/trends)
Air gaps (the good old? times)

Not completely secure though

- Infected USB pendrives (eg: Stuxnet attack)
- Insider attacks (unhappy employees, bribery, blackmail..)
- Attacks to the source code repositories
- Breaking into local Wifi networks through smartphones
  - or drones!
Going IoT (energy optimization)

Data Source: Google Trends (www.google.com/trends)

- Boiler
- Actuator
- Temperature sensor
- Boiler heater
- Firewall
- Power station
- Gateway
- Control and data centre
- Embedded device
- Thing

Facility-side

User-side

Home energy management solution

- Air conditioner
- LED lighting
- Battery
- Tablet WiFi
- Bluetooth
- Home gateway
- Energy measurement unit
- Cloud
- Router
- LAN
What we want to protect

Information security
- Authentication, integrity, confidentiality, availability..
  - Identity theft, privacy leaks, falsified energy usage..

Security impact on Safety
- Protect the “Things”
  - Nature, human lives, infrastructure, energy, equipment..

Source: US Department of Homeland Security

2007: Attack to the US power grid (industrial turbine spinning wildly out of control)
Facility-side embedded devices

Requirements

- Safety and high reliability
- Real-time response guarantees
- Software certification (tests, formal methods, ..)
- Continuous operation
- Fast booting
Practical constraints

- **Real-time requirements**
  - Weak to disturbances (DoS attacks)

- **Updating and re-certifying embedded software is costly**
  - Certified legacy software (~20 years untouched).
  - Rebooting can be expensive or dangerous (heating controller)

- **Fast booting**
  - Difficult to make it compatible with security booting

- **Low performance devices**
  - Some security countermeasures might cause too much overhead

- **Hardware-assisted security varies with the board**
  - Cortex-M3, Cortex-A9, PPC, SH, x86, x86_64..
(My) Three key security guidelines

1. **Reduce the attack surface**
   - Remove *anything* that is not used (not just restrict it to root)
   - Do you really need the ptrace system call?
     - or the kernel symbols, or modules, or gdb…

2. **Leverage the determinism of your system**
   - Look for anomalies that were supposed not to occur
     - Allows for security solutions that generalize to many attacks.
   - Example
     - Prevent new processes from being created in a real-time system.
     - Check the amount of network connections.

3. **Isolate critical software from less trustable software**
   - Reduce the impact of successful attacks
Kernel security hacking for the IoT

1. Introduction
2. Reducing the attack surface
3. Leveraging determinism
4. Protecting the critical software
5. Conclusions
Remove anything unused

My point

- Unused interfaces are often the most vulnerable.
- Attackers usually go for the lower hanging fruit.

Kernel

- System calls: ptrace, process_vm_write, iopl, _sysctl …
  - Harden the needed ones: mprotect (Grsecurity)
- Information leaks: kallsyms, proc, sys, debugfs, kprobes…
- Kernel trojans: /dev/kmem, modules, kexec, ksplice, …

File system customization

- RO filesystem with remounting disabled
- Don’t install tools that are useful for attackers (unless required)
  - Objdump, perl, apt-get, mkfs, reboot
Use case: removing unused system calls

System calls

- The Linux kernel source code is complex and grows every minute.
- Commonly used system calls are reasonably secure
  - Except those aimed at debugging, such as ptrace
- But rarely used or recently introduced ones often contain bugs that may lead to security problems.
How to get rid of them

**Step 1: syscall identification**
- Tracing the application: see `./trace-syscalls.sh`
- Extract library calls (see `libc-parser.py`) and map them to syscalls
- `find-syscalls.py`: [https://github.com/tbird20d/auto-reduce](https://github.com/tbird20d/auto-reduce) (by Tim Bird)

**Step 2: syscall removal**
- Modify the kernel system call table (see below).
- Kernel tinification: [https://tiny.wiki.kernel.org/sysevents](https://tiny.wiki.kernel.org/sysevents)
- Tim Bird patches: [http://elinux.org/System_Size_Auto-Reduction](http://elinux.org/System_Size_Auto-Reduction)

```bash
$ vi arch/x86/syscalls/syscalltbl.sh
  - linkat sys_linkat
  + linkat sys_ids_syscall
$ vi hello.c
  ret = linkat(AT_FDCWD, "hacker.txt", AT_FDCWD, "/etc/passwd", 0);
  if (ret != 0) perror("linkat");
$ ./hello.exe
  linkat: Function not implemented.
```

The system call was not executed. Optionally, we can be stealthy and return no error.
Evaluation

Percentage of system call attack surface reduction

- Simple applications such as ‘ls’ or ‘tcpdump’ only used about 30 unique system calls in average.
- For x86, which has ~350 system calls, that represents a 91% reduction of the syscall attack surface.
Using seccom-bpf

Seccom-bpf (SECCOMP_SET_MODE_FILTER)

System calls disabled per application

application A

application B

System calls not used by the system

Linux

```
struct sock_filter filter[] = {
    ALLOW_SYSCALL(rt_sigreturn),
    ALLOW_SYSCALL(exit),
    ALLOW_SYSCALL(read),
    ALLOW_SYSCALL(write),
    ALLOW_SYSCALL(close),
    ...
};
struct sock_fprog prog = {
    .len = (unsigned short)(sizeof(filter)/sizeof(filter[0])),
    .filter = filter,
};
prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0)
prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, &prog)
ret = syscall(69);
printf ("should not arrive here\n");
```

See bpf_syscall_error.c
There is more information we can use

**Firefox (complex application)**

- Note that the frequency depends greatly of the system call executed. This and other information can be used to refine the mechanism furthermore.
Kernel security hacking for the IoT

1. Introduction
2. Reducing the attack surface
3. Leveraging determinism
4. Protecting the critical software
5. Conclusions
Anomaly-based intrusion detection/prevention

Overview

- Leverage the determinism of your embedded systems
  - Detect anomalies that divert from expected behavior

What determinism?

- Task periods, maximum IRQs/s, task’s CPU time per period
- Device accesses: timing, order, allowed tasks
- Fixed number of processes
- Process sections’ (text, GOT table) hashes
- Files accessed by each application
- Processes crashes shouldn’t happen
- Network: connections, packet patterns, packet sizes..
HIDS: Host-based intrusion detection systems

**Syscall-based HIDS**

- Track the execution of the system calls used by an application
  - Look for anomalies (e.g., syscall order, arguments, timing)
  - Small bound CPU overhead expected on the target application

```plaintext
1: open()
2: read()
3: setreuid()
4: mmap()
5: open()
6: write()
7: mmap()
```

Normal execution sequence

```plaintext
1: open()
2: read()
3: setreuid()
4: mmap()
5: open()
6: write()
7: mmap()
```

Execution sequence after a stack overflow or ROP attack

```plaintext
1: open()
2: read()
3: setreuid()
4: mmap()
5: open() 3: mprotect()
6: write() 4: mmap()
7: mmap() 5: write()
```
System call monitor (proof of concept)

- **Extraction phase**: Extract the system call sequence of the target application

- **Initialization phase**: Set up the security settings when starting the target application

- **Execution phase**: Check that the target application doesn’t try to execute an unexpected system call

- When an unexpected system call occurs, communicate it to the Audit subsystem and kill the app or change it to fail safe mode

Command:

```
# ./trace-syscalls.sh \ command <args>
```
Execution phase

Monitoring

During execution the system calls called by the target application need to be checked. This task is performed inside the kernel.

- See 0002-syscall-hids-proof-of-concept-version-of-a-syscall-h.patch

```
system call trap vector for x86 arch
entry_32.S:syscall ioctl

stub function shared With seccomp
syscall_trace_enter()

check that the system call execution order is as expected
__secure_computing()

Previous syscall               Next possible sysscalls
epoll_wait                    ioctl         socketcall      read
sendfile64                    close         time            epoll_ctl
setsockopt                    ioctl         fcntl64
```

since the previous call was ‘setsockopt’, the ‘ioctl’ is allowed execution

prev_syscall = setsockopt
Anomaly detection HIDS map

About 2 decades of papers

Integrity

Secure booting
- ROM → Bootloader → Kernel → Modules

File system integrity
- AIDE
- Linux IMA/EVM
  - Check file and metadata integrity when the application is started

Problems:
- One-time checks
  - Rebooting devices or RT apps in a power station is not safe
- React after the damage is done (prevention is best)
- Does not address modifications to the process memory
  - There are many ways to do that (even with DEP)
Inotify-based file integrity monitoring

- Simple script that can be extended

```python
import pyinotify
import sys
from mailer import send_sfm_report
wm = pyinotify.WatchManager()
mask = pyinotify.IN_ACCESS | pyinotify.IN_ATTRIB

class EventHandler(pyinotify.ProcessEvent):
    def process_default(self, event):
        send_sfm_report(event.pathname)

handler = EventHandler()
notifier = pyinotify.Notifier(wm, handler)
wdd = wm.add_watch(sys.argv[1], mask, rec=True)
notifier.loop()
```

- Other file operations to check
  - IN_CREATE, IN_OPEN, ...

- Check for things that shouldn’t happen
  - This way we can get security with no overhead in the common case

See sfm.py
Attack to a memory resident app

**Integrity of `.text/.got/.got.plt` data**

- mprotect, GOT, buffer overflow attacks
- file integrity vs. memory integrity

**Before attack**

```
line 31st
```

```
80481d: b5 c0     test  %eax,%eax
80481cf: 74 0c     je  80481dd <_do_global_dtors_aux+0x5d>
80481d1: c7 04 24 98 03 0c 08  movl  $0x80c0398,(%esp)
80481d8: e8 f3 e4 05 00  call  80a56dd <_deregister_frame_info>
80481dd: c6 05 a0 69 0c 08 01  movb  $0xl,0x80c039a0

80481e4: 83 c4 14  add  $0x14,%esp
80481e7: 5b        pop  %ebx
80481e8: 5d        pop  %ebp
80481e9: c3        ret
80481ea: 8d b6 00 00 00 00  lea  0x0(%esi),%esi
```

**After attack**

```
80481e8: b8 d0 66 0a 08  mov  $0x80a66d0,%eax
80481ed: b5 c0     test  %eax,%eax
80481cf: 74 0c     je  80481dd <_do_global_dtors_aux+0x5d>
80481d1: c7 04 24 98 03 0c 08  movl  $0x80c0398,(%esp)
80481d8: e8 f3 e4 05 00  call  80a56dd <_deregister_frame_info>
80481dd: c6 05 a0 69 0c 08 01  movb  $0xl,0x80c039a0

80481e4: 90        nop
80481e5: c4 14 5b  les  [%ebx,%ebx,2],%edx
80481e8: 5d        pop  %ebp
80481e9: c3        ret
80481ea: 8d b6 00 00 00 00  lea  0x0(%esi),%esi
```

'add' becomes 'nop'
Kernel integrity monitor (prototype)

Monitor flow chart

- Kernel thread running periodically in background

Note: XOR should be changed to a better hash algorithm
Kernel security hacking for the IoT

1. Introduction
2. Reducing the attack surface
3. Leveraging determinism
4. Protecting the critical software
5. Conclusions
Linux partitioning

Containers
- Core isolation for the real-time performance of critical software
- Restrict the amount of resources that less trustable software can use
- Device cgroups: only block and character devices
  - See 0001-cgroups-devices-add-experimental-support-for-network.patch
Hardware-assisted architecture

SafeG (Nagoya University)

- Allows running an RTOS and Linux in parallel (single and multi-core)
- Protection against peripheral DMA attacks.
- Get it!
  - https://www.toppers.jp/safeg.html (日本語)
  - Latest: https://www.toppers.jp/download.cgi/safeg-1.0.tar.gz

Source: https://www.toppers.jp/safeg.html
Kernel security hacking for the IoT

1. Introduction
2. Reducing the attack surface
3. Leveraging determinism
4. Protecting the critical software
5. Conclusions
(My) Three key security guidelines

1. **Reduce the attack surface**
   - Remove anything that is not used (not just restrict it to root)
     - System call removal
     - Seccomp filter

2. **Leverage the determinism of your system**
   - Look for anomalies that were supposed not to occur
     - System call based kernel-level intrusion detector
     - File integrity monitor
     - Process memory integrity checker (kernel module)

3. **Isolate critical software from less trustable software**
   - Reduce the impact of successful attacks
     - Cgroup device kernel patch
     - SafeG (TrustZone monitor implementation)
A few things I didn’t talk about

Cloud or user-side device’s security
- Focus on the safety of the embedded devices at the “facility-side”
  - Eg: civil infrastructure systems (power, water, transport..)

Network security
- Cryptography, authentication, gateway, firewalls, NIDS (Snort)…

Access control
- Permissions, capabilities, suid, SELinux

Traditional anti-virus
- Focus on anomaly-based attack prevention systems

Hardening
- CFLAGS += "-fstack-protector -pie -fPIE -Wl,-z,relro -Wl,-z,now"
- checksec.pl
Future topics

- **Community software quality improvements**
  - Bug bounty programs, peer-reviews, formal methods..

- **Incident response**
  - What if secure booting detects a problem?

- **Attribution (tracking down the attackers)**

- **Coordinated node blacklisting**
  - Blacklist stolen or compromised nodes.

- **Stackable LSM (Linux Security Modules) and Seccomp**
  - Incompatibilities can be defined at Kconfig level

- **Safe and secure dynamic update technology**

- **Generic solutions (one ring to rule them all)**
On-going work

**Simplify embedded security deployment**

- We need to automatize know-how, patterns and best practices
  - Meta-security: kernel settings, busybox configuration, security tests (RIPE, checksec.pl, metasploitable, fuzzy), strip binaries..

**Understand what your system is running**

- RTOS developers are used to know everything the system has!
- Make it easy to identify all inputs, attack surface
- My small script: deadfile eliminator

See deadfile_eliminator-*\.py (application agnostic)

<table>
<thead>
<tr>
<th></th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>8079</td>
<td>45653</td>
</tr>
<tr>
<td>U-boot</td>
<td>427</td>
<td>7175</td>
</tr>
<tr>
<td>QEMU</td>
<td>1321</td>
<td>7794</td>
</tr>
</tbody>
</table>
Thanks for your attention

Proof of concept code: