How to develop a rootkit for Broadcom NetExtreme network cards

Guillaume Delugré
Sogeti / ESEC R&D
guillaume(at)security-labs.org

Recon 2011 - Montreal

Plan

1 Reverse engineering NetExtreme firmware

2 Rootkit architecture

The targeted device

Previous work

Targeted hardware

- Broadcom NetExtreme family of Ethernet cards
- Widely used on laptops, home computers, servers...
- MAC components
  - MIPS CPU (executing firmware)
  - Non-volatile EEPROM memory (firmware, config...)
  - Volatile SRAM memory (code, stack, packet buffers...)

Previous work

- Developed a set of tools to reverse engineer NetExtreme firmware
- Full access to ROM, EEPROM and SRAM in software
- Debugging of the firmware in real time from userland
- EEPROM write access, code execution at bootstrap

InVivo debugger interface (powered by Metasm)

```
File  Actions  Options  Views

lw $a1, 6324h($a1)
lui $v0, -102h
ori $v0, $v0, 1bh
sw $ra, 20h($sp)
sw $s3, 1ch($sp)
sw $s1, 14h($sp)
sw $s0, 10h($sp)
sw $v0, 0bch($v1)
lw $v1, 1dch($a1)
lui $v0, 1
ori $v0, $v0, -4b48h
addiu $a0, $a0, 0dch
lui $at, 1
sw $v0, 6378h($at)
li $v0, -8000h
lui $at, 1
sw $a0, 6328h($at)
andi $v1, $v1, -100h
bne $v0, $v1, loc_10c70h ; x:loc_10c70h
nop

lui $v0, 1
lw $v0, 6324h($v0)
lbu $v0, 1dfh($v0)
bnz $v0, loc_10c70h      ; x:loc_10c70h
nop

lui $v0, 1
lw $v0, 6324h($v0)
lw $v1, 0($v0)
lui $v0, -2153h
ori $v0, $v0, -2153h
beq $v0, $v1, loc_10c70h ; x:loc_10c70h
nop

lui $v1, 1
lw $v1, 6324h($v1)
lui $v0, -4b66h
ori $v0, $v0, -7655h
sw $v0, 0($v1)

// Xrefs: 10c24h 10c38h 10c54h
loc_10c70h:
sw $zero, 6810h($gp)
lw $v0, 6c20h($gp)
lui $s3, 2
bz $v0, loc_10c90h          ; x:loc_10c90h
lw $s3, -5fd8h($s3)

lw $v0, 6c00h($gp)
j loc_10c98h                ; x:loc_10c98h
ori $v0, $v0, 9feh

// Xrefs: 10c7ch
loc_10c90h:
lw $v0, 6c00h($gp)
ori $v0, $v0, 9fch

// Xrefs: 10c88h
loc_10c98h:
jal sub_14790h              ; x:sub_14790h
sw $v0, 6c00h($gp)
jal sub_145f0h              ; x:sub_145f0h
```

Firmware development

Firmware development

- Firmware developed in C, very few MIPS assembly required
- Replacing the proprietary Broadcom firmware ?
- Also a nice place to hide a rootkit
    - Remote access for an attacker
    - Access to physical memory over PCI

Plan

1 Reverse engineering NetExtreme firmware

2 Rootkit architecture

So what now?

Thinking of a plan

- Now we can load and execute firmware code in the NIC
- Where do we go from here?

Common beliefs

- The firmware is responsible for processing network frames
- The firmware controls everything in the NIC

So what now?

Thinking of a plan

- Now we can load and execute firmware code in the NIC
- Where do we go from here?

Common beliefs

- The firmware is responsible for processing network frames
- The firmware controls everything in the NIC

WRONG!

In real life

Here is the real story

- The firmware initializes the NIC (low-level stuff)
- It would possibly handle special features (WoL, PXE...) if requested
- Once the NIC is ready to be used by driver, the firmware stalls its execution
- **The End**.

In a nutshell

- Firmware almost does nothing
- Much of the work is directly inlined in hardware

In real life

Here is the real story

- The firmware initializes the NIC (low-level stuff)
- It would possibly handle special features (WoL, PXE...) if requested
- Once the NIC is ready to be used by driver, the firmware stalls its execution
- **The End**.

In a nutshell

- Firmware almost does nothing
- Much of the work is directly inlined in hardware

So what now?

The challenge

- So we got code execution on something which does nothing
- How can we turn this something to really do something?
- That is the real topic of this presentation

Requested features

- We want to communicate remotely with the firmware
- We want to do arbitrary DMAs to physical memory
- Need to gain control over the Ethernet link and the DMA engines from firmware

Plan

1. Reverse engineering NetExtreme firmware
2. Rootkit architecture
   - Firmware initialization
   - Intercepting the network flow
   - Hacking the DMA engines

Firmware initialization

Two options

1. Let the firmware run and hook its execution flow afterwards
2. Rewrite the whole firmware initialization sequence

Hooking the firmware execution flow

Hooking the firmware execution flow

- Much more easy and efficient
- It is possible without modifying the firmware code/data
- Original firmware can optionaly load other firmware
- Just burn the firmware and a special flag in the EEPROM
- The original firmware will automatically load yours after init

Just a little thing to take care of

- You cannot directly rewrite the code which is loading the firmware (it will obviously crash)
- Load a trampoline code at a higher address
- Jump on it and rewrite the original firmware from there

Rewriting the whole firmware bootstrap routine

Rewriting the firmware initialization sequence

- If you feel sport...
- Quick way: trace all the I/O accesses and replay them
- I started reversing the whole thing, but it uses a lot of undocumented registers

Fun fact

- The EEPROM header is CRC32-checked by ROM code
- The ROM code for BCM5721 is bugged
- Firstly, CRC32 algorithm is wrongly implemented
- Performs CRC over the whole header, including the CRC field
- Compares with a static constant inlined in the assembly code
- Bruteforce the CRC32 field until it matches the constant...

Plan

1. Reverse engineering NetExtreme firmware
2. Rootkit architecture
   - Firmware initialization
   - Intercepting the network flow
   - Hacking the DMA engines

Intercepting the network flow

Objective

- We want to establish a communication channel with a remote attacker
- But the default firmware cannot do that...

ASF

- *Alert Standard Format*
- A special firmware from Broadcom is dedicated to remote management
- System can be halted/rebooted remotely
- SNMP heartbeats are regularly sent over the network
- How does it do that ?!

ASF firmware

Reversing the ASF firmware

- This Broadcom firmware is actually a complete hack
- Makes use of an integrated QoS feature to catch special packets (ASF)
- Makes use of a poorly documented register to forge network frames (SNMP heartbeats)

QoS in network card

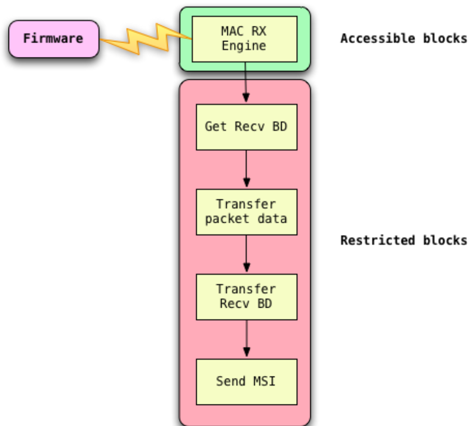
Quality of service

- A few models of cards supports integrated QoS
- Incoming packets are placed into different queues depending on their data
- Although most models supposedly don't support it, the registers are still here

Mechanism

1. Up to 8 rules can be set up (logical conditions)
2. If a packet matches a rule, the RX flow is halted
3. An interrupt is sent to the MIPS CPU
4. Firmware can then inspect packet or even modify it
5. Packet can be dropped or passed to the OS

Hooking the RX flow



Forging packets

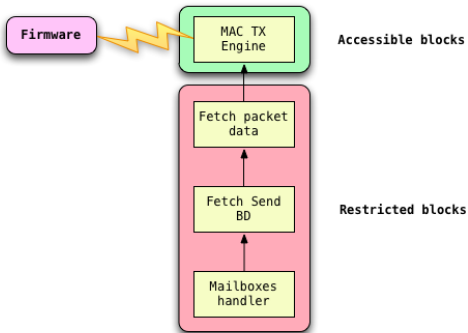
How to send packets from the firmware?

- ASF firmware makes use of a poorly documented register
- Some undocumented structures are set up in NIC memory
- Those structures contain chunks of packet data
- A pointer to the structures list is put into the register
- The packet magically appears on the network!

Reproducing the trick

- The thing was to understand those structures
- I found very similar ones defined in a very old driver C header
- They were never referenced in the code, no reason to be here
- Probably a wrong copy-paste from Broadcom developers. . .

Forging packets



Summary at this point

Communication over the network

- So now I can catch any incoming packet. . .
- . . . and I can also forge new ones
- Consequently I can remotely communicate with the NIC

Can we also intercept the TX flow?

- The answer is no, at least not easily
- No similar QoS thing exists for the outgoing packets
- The driver can request TCP segmentation off-loading by firmware
- But no driver actually does that. . .
- We will come back to this feature later however

Final step for network processing

Networking setup

- Firmware only knows its own MAC address
- We need the gateway MAC, our own IP, maybe the DNS server IP...

Detecting the network configuration

- Rootkit has its own IP/UDP stack
- Embeds a lightweight DHCP client
- If no DHCP server on LAN, catches DNS responses and extract MAC gateway, our own IP, DNS IP...
- A fake MAC address can also be used to simulate a ghost machine on the LAN

Plan

1. Reverse engineering NetExtreme firmware
2. Rootkit architecture
   - Firmware initialization
   - Intercepting the network flow
   - Hacking the DMA engines

Getting DMAs to work

Next step!

- We can communicate with the firmware remotely. Cool.
- But it would be quite pointless if we couldn't attack the OS
- For this, we need to be able to do arbitrary DMAs to physical memory

DMA

- Packets are exchanged between the driver and the NIC through DMA over PCI
- Everything is automatized, firmware does not interfere
- To hack the DMA engines, we need to delve into the NIC internals

NIC internals

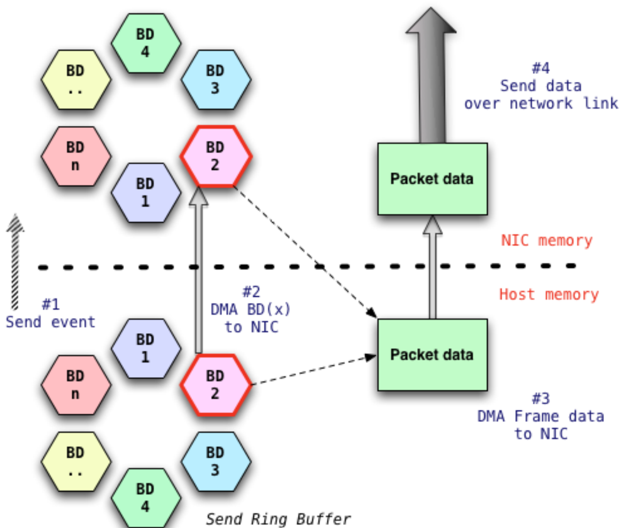
Exchanging packets between NIC and the driver

- A buffer is described by a *Buffer Descriptor* structure (BD)
  - Pointer to data in physical memory
  - Length of data
  - Some optional flags for the NIC
- Buffer descriptors are organized into *Ring Buffers*
- NIC and driver use a consumer/producer design pattern

Example: Packet send (TX)

- Here, driver = producer and NIC = consumer
- Driver produces a BD, increments producer index
- NIC fetches the pending BD, then packet data
- NIC increments its consumer index, and so on

TX process



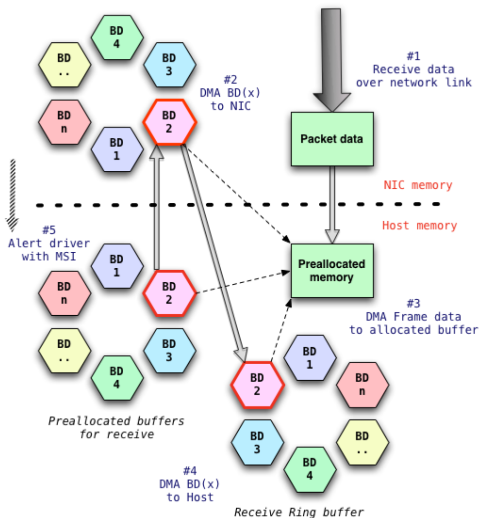
NIC internals

Example: Packet reception (RX)

- RX process is a little more complicated, uses two rings
  - One with BDs pointing to preallocated memory
  - One with BDs pointing to packet data
- Driver produces BDs with preallocated memory
- NIC consumes those BDs when a packet is received
- NIC copies packet in preallocated memory and produces a new BD
- Driver consumes the BD and finds the packet data in memory

Producer and consumer indexes for each ring must be consistent on the driver and NIC sides

RX process



Status block

Status block

- The NIC keeps track of its indexes into a *Status Block* structure
- The NIC regularly DMAs this structure to the driver
- Driver checks
  - if some packets are pending for reception
  - if sent packets have been properly processed (then it can free their memory)

Subverting the NIC

DMA

- Buffer descriptors point to physical memory
- To achieve arbitrary DMAs, we need to corrupt *Buffer Descriptors* just in time
  - For write access, corruption of BDs in the RX ring
  - For read access, corruption of BDs in the TX ring
- Strong constraints
  - BDs have to be corrupted just in time
  - NIC and driver must keep working normally
- We will begin with the DMA writes which are much simpler to achieve

DMA write

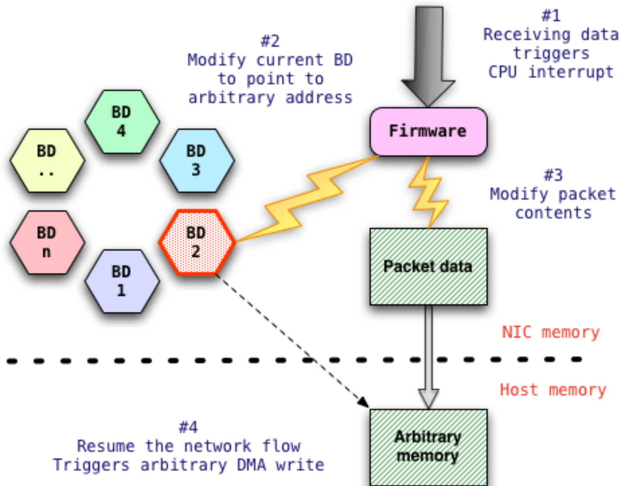
Principle for DMA write

- We can interrupt the RX flow with the QoS trick
- Firmware corrupts the current BD in NIC memory
  - Makes it point to an arbitrary address in physical memory
- Firmware modifies the packet contents in NIC memory
- Firmware then resumes the RX flow

Impact

- Arbitrary data is then DMAed to arbitrary address
- Driver sees an invalid packet, but keeps on working
- We can write between 4 and 1500 (MTU) bytes to physical memory in one shot

DMA write



DMA read

DMA read

- DMA read is much more difficult to achieve, but still feasible
- The problem is we cannot interrupt the TX flow as for RX
- So we will
  - simulates a TX event from firmware
  - makes it trigger an interruption to the MIPS CPU
- **Everything will be done using only DMA writes**

TCP segmentation feature

- TX process can be interrupted if a special flag is set in the BD
- Firmware can then offload TCP segmentation from the host CPU
- Driver never sets this flag by default, so we'll do it ourselves

DMA read

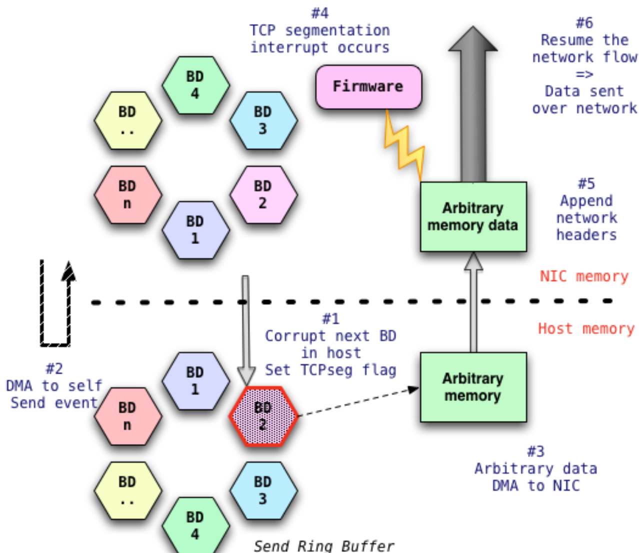
Principle for DMA read

- Corrupt the current BD in TX ring with **DMA write**
  - Modify the address/length of data
  - Set the TCP segmentation bit
- Trigger a TX event
  - Register is not accessible from the MIPS CPU
  - Emits a **DMA write** to the NIC itself to bypass that
  - We can do this because we have access to the PCI BAR0

Impact

- Arbitrary memory is DMAed to the NIC
- TCP segmentation interrupts the MIPS CPU
- Firmware appends network headers to data, resumes TX flow
- Arbitrary data is sent over the wire

DMA read



DMA read

We are not done yet

- After experimentation, a packet with physical memory does appear on network
- HURRAY!
- Then the driver miserably crashes...

What happened?

- The NIC has consumed a BD the driver has never produced
- We have `Index_NIC == Index_driver + 1`
- Driver does see this, enters some recovery routine which crashes the whole thing
- **We must resynchronize the driver with the NIC**

Resynchronization technique #1

Resynchronization technique #1

- Driver won't crash if we prevent the Status Block from being DMAed
- The ring buffer is usually made of 512 BD entries
- We use the same technique to trigger 512 DMA reads in a row
- At this time we have gone around the full ring to our initial point
- We can resume Status Block updating, driver and NIC are now resynchronized

Drawback

- Triggers 512 DMA reads, quite a waste if we just want a few bytes

Resynchronization technique #2

Edge case behaviour

- We actually have `Index_NIC == Index_driver + 1`
- If we block the Status Block update, driver will still see `Index_NIC == Index_driver`
- So if a new packet is sent, driver will do `Index_driver + 1`, which equals to `Index_NIC`
- Nothing happens on the NIC side, but the driver is resynchronized
- We can try to force the driver to send a packet (which will never be processed)

Idea

- Fake the reception of an ARP request packet to the driver
- ARP reply will be automatically sent

Resynchronization technique #2

Faking a packet reception

- We have to fake the arrival of the ARP request, but we cannot let the Status Block being updated (or the driver will crash)
- Methodology:
  - Block the Status Block updating process
  - Replace a packet data with a fake ARP request packet
  - Inject a fake Status Block in physical memory (RX index incremented)
  - Send an interrupt to the driver to fake the packet arrival
  - ARP reply is sent, discarded, and driver is resynchronized
  - Restore the Status Block updating process

Resynchronization technique #2

Faking a packet reception

- We have to fake the arrival of the ARP request, but we cannot let the Status Block being updated (or the driver will crash)
- Methodology:
  - Block the Status Block updating process
  - Replace a packet data with a fake ARP request packet
  - Inject a fake Status Block in physical memory (RX index incremented)
  - **Send an interrupt to the driver to fake the packet arrival**
  - ARP reply is sent, discarded, and driver is resynchronized
  - Restore the Status Block updating process

Resynchronization technique #2

Triggering an interrupt from firmware

- If standard interrupts are used, then assert the PCI `#INTA` line
- If MSI are used (e.g. over PCI-Express) then
  - Emits a DMA write to the APIC at `0xffe0xxxx`
  - This will trigger a *Message Signaled Interrupt* (MSI)
  - We know the exact address and vector to use since it has been assigned to us by the chipset
  - This information can be found in the PCI configuration space
  - APIC will then send the right IRQ to the driver (no spurious interrupt)

Advantage

- After all this, we have triggered only **one** DMA read
- This technique is much more hardcore, but it spares us from triggering 512 DMA reads in a row

Conclusion on DMA

Bypassing inherent limitations

- Firmware was initially not meant to control the DMA engines
- DMA writes are yet achievable in a very stable way
- At first glance DMA reads looked unfeasible...
- The presented techniques make it possible using only DMA writes at predictable addresses
- No configured IOMMU was supposed to be present

Remote DMA

- Rootkit offers read/write primitives to physical memory to the remote attacker
- Small and independent from the underlying operating system
- Intelligence has to reside on the client side

Conclusion

In a nutshell. . .

- Getting code execution on the NIC is **NOT** the end of the game
- A lot of trickery is needed to turn the firmware into a malicious payload
- Rootkit can communicate with the attacker over the network
- R/W access primitives to physical memory are provided

Security considerations

- Rootkit is active even if the machine is shutdown
- Rootkit can corrupt physical memory during the system boot
- Need for a trusted boot sequence coupled with a properly configured IOMMU

References

References

- My talks about reverse engineering the Broadcom firmware (Hack.lu 2010, HITB 2011)
- *Can you still trust your network card*, Y-A. Perez, L. Duflot (CSW 2010)
- *Runtime firmware integrity verification*, Y-A. Perez, L. Duflot (CSW 2011)

Thank you for your attention!

Questions?