fuzzing & exploiting wireless device drivers

Vienna, 23 November 2007

Sylvester Keil
sk (at) seclab (dot) tuwien (dot) ac (dot) at

Clemens Kolbitsch
ck (at) seclab (dot) tuwien (dot) ac (dot) at
Agenda

• 802.11 fundamentals
• 802.11 fuzzing
• Virtual 802.11 fuzzing & live demonstration
• Kernel-mode exploits primer
introduction
About us

• We are two students from the Technical University Vienna

• Right now we ought to be working on our master theses at the Secure Systems Lab @ TU Vienna

• The work presented here is based on the results of a seminar paper we wrote during a collaboration between the Secure Systems Lab and SEC Consult

• SEC Consult also has a “Vulnerability Bonus Program” – for details see http://www.sec-consult.com or mail to vulnerabilities@sec-consult.com
Wireless networks have become a widely used means of communication. Compatible devices are included in most portable computers, mobile phones, etc.

That means, there is an increasing number of mobile targets out there...

What’s more, the device drivers typically operate in supervisor-mode (i.e. in kernel-space), thus rendering vulnerabilities extremely dangerous.
The IEEE 802 Family

802 Overview and architecture

802.1 Management

802.2 Logical link control (LLC)

802.3 MAC

802.3 PHY

802.11 Media access control (MAC)

802.11 PHY

802.11 PHY

802.11 PHY

Data Link Layer

Physical Layer
Three types of frames: **management**, **control** and **data frames**

Management frames used to advertise and connect to networks.
802.11 states

- State 1
  - Class 1
- State 2
  - Classes 1, 2
- State 3
  - Classes 1, 2, 3

- Authentication
- Association or Reassociation
- DeAuthentication
- Disassociation
802.11 association

access point

station
802.11 association

access point

Beacons

station
802.11 association

1. Beacons

2. Probe Request

access point

station
802.11 association

1. Beacons

2. Probe Request

3. Probe Response

access point

station
802.11 association

1. Beacons
2. Probe Request
3. Probe Response
4. Authentication

access point

station

DEEPSEC 2007
802.11 association

1. Beacons

2. Probe Request

3. Probe Response

4. Authentication

5. Authentication
802.11 association

1. Beacons
2. Probe Request
3. Probe Response
4. Authentication
5. Authentication

access point  

State 2  

station
802.11 association

1. Beacons
2. Probe Request
3. Probe Response
4. Authentication
5. Authentication
6. Association Request
802.11 association

1. Beacons

2. Probe Request

3. Probe Response

4. Authentication

5. Authentication

6. Association Request

7. Association Response
802.11 association

access point

1. Beacons
2. Probe Request
3. Probe Response
4. Authentication
5. Association Request
6. Association Response
7. State 3

station
802.11 fuzzing
802.11 fuzzing issues

- Fuzzers must be aware of frequency channels, BSSIDs, states, modes, and data link encryption (filtering may take place at hardware level!)

- **Response time and timing** of replies is critical (e.g., because of reply windows or channel hopping)

- Overload, interference, packet corruption may occur

- Attacker and target must be **co-ordinated** and target must be continuously monitored
What to fuzz?

• Some Information Elements (IE) follow type-length-value pattern

• Type and length fields have fixed size, the value field’s size is variable (potential overflow)
Example: a beacon frame

- **Time-stamp**
  - ID: 0x0
  - LEN: 0x9
- **Beacon Interval**
  - SSID: 'MyNetwork'
  - Supported Rates: 11.0 (B) → 54.0 (B)
- **Capability Information**
  - ID: 0x1
  - LEN: 0x8
  - Freq: 0x9

**Frame Control (FC)**
- ID: 0x00

**Destination and Source**
- ID: 0x9

**BSSID**
- ID: 0x8

**Frame Check Sequence (FCS)**
virtual 802.11 fuzzing
A novel approach

- **Requirements**
  - Eliminate *timing constraints*
  - Replace *unstable wireless medium*
  - Allow *guaranteed delivery*
  - Support *advanced target monitoring*

- **Solution**
  - *Move target into a virtual environment!*
Advantages

- **Virtual wireless device** (software) replaces network hardware
- **High-level IPC** instead of live frame-injection
- CPU of virtual machine can be interrupted and stopped if necessary
- Guest OS monitoring at low-level (system restart, console output, etc.)
- Drastically simplifies complexity of fuzzing process
Our solution

• Develop a fuzzing “framework” on the basis of Fabrice Bellard’s QEMU (optional ethernet card can be added via command-line option)

• Modular design
  • packets read from outgoing queue are copied to shared memory
  • connected modules are notified via semaphores
  • packets are read from shared memory and copied to incoming queue
System overview

**QEMU**

- CPU
- MMU
- Ethernet

**802.11 Fuzzer**

- Shared memory
  - Reply (RM)
  - Inject (IM)

**PCI ID**: 168c0013 (rev01)
Atheros Communications, Inc.
AR5212 802.11abg NIC (rev 01)

**Dumper (RM)**: store outgoing packets

**Listener (RM)**: display outgoing packets

**Injector (IM)**: inject arbitrary packets

**Stateless fuzzer (IM)**: reply directly

**Access point (RM & IM)**

**Stateful fuzzer (RM & IM)**: AP and fuzzer
Access Point module

- Broadcasts **beacon** frames
- Responds to incoming **probe requests**
- Supports complete **Open System Authentication**
- Responds to incoming **association requests**
- Features minimum implementation of **ICMP**
- Full **logging of 802.11 traffic**
- But words can only say so much…
Stateful fuzzer module

• Initially, the fuzzer behaves like an access point module, broadcasting valid beacons and responding to probe requests.

• Once authentication is complete, it is possible to fuzz the target in state 2, e.g. transmit fuzzed association response frames.

• See it yourself…
fuzzing results
Results

- We have developed a “framework” for 802.11 fuzzing using QEMU
- So far the framework supports fuzzing in all three states of a target in managed mode
- A simple fuzzer using the framework and old versions of the MadWifi driver detected known vulnerabilities
- A previously undocumented vulnerability was also found!
The vulnerability

- Our fuzzer detected a flaw in the MadWifi implementation
- A **beacon** frame with a specially crafted **Extended Supported Rates** information element crashes Linux when scanning for available networks
- Sadly (uh, is deepsec blackhat?), no remote code execution possible (but DoS)
- Recently published by **SEC Consult & TU Vienna** and fixed since 0.9.3.3

DEEPSEC 2007
kernel-mode exploits
Vulnerabilities in kernel space
Vulnerabilities in kernel space

• What types of kernel space vulnerabilities are there?
Vulnerabilities in kernel space

- What types of kernel space vulnerabilities are there?
- How can they be exploited (remotely)?
Vulnerabilities in kernel space

- What types of kernel space vulnerabilities are there?
- How can they be exploited (remotely)?
- How generic are these exploits?
NULL / user-space dereference
NULL / user-space dereference
foo = kmalloc(size, GFP_KERNEL);
/* if kmalloc fails, foo will be NULL */
...
/* later on... */
foo->data->value = some_value;
NULL / user-space dereference

```
... 
foo = kmalloc(size, GFP_KERNEL);
/* if kmalloc fails, foo will be NULL */
...
/* later on... */
foo->data->value = some_value;
```
NULL / user-space dereference

... foo = kmalloc(size, GFP_KERNEL);
/* if kmalloc fails, foo will be NULL */
...
/* later on... */
foo->data->value = some_value;
Heap (slab) overflows
Heap (slab) overflows

- The slab allocator can create so called lookaside caches for you: pools of memory objects (slabs) that all have the same size.
Heap (slab) overflows

- The **slab allocator** can create so called **lookaside caches** for you: pools of memory objects (**slabs**) that all have the same size.

- **kmalloc** uses just a number of such pools!
Heap (slab) overflows

• The slab allocator can create so called lookaside caches for you: pools of memory objects (slabs) that all have the same size.

• kmalloc uses just a number of such pools!

• A slab overflow is, if we write beyond the boundary of a slab and into the adjacent slab.
Heap (slab) overflows

- The **slab allocator** can create so called **lookaside caches** for you: pools of memory objects (**slabs**) that all have the same size.
- **kmalloc** uses just a number of such pools!
- A **slab overflow** is, if we write beyond the boundary of a slab and into the adjacent slab.
Heap (slab) overflows

- The slab allocator can create so called lookaside caches for you: pools of memory objects (slabs) that all have the same size.

- kmalloc uses just a number of such pools!

- A slab overflow is, if we write beyond the boundary of a slab and into the adjacent slab.

- If we know the contents of the adjacent slab, we might be able to overwrite a pointer and thus create a pointer dereference exploit, or similar scenario.
Stack overflows

- Typically, kernel stack is 4k or 8k.

- Otherwise similar to user stack exploits: overwrite saved return address with buffer address.

- How do we know where to jump to? And how do we know the location of the saved return address?
Stack overflows

• Typically, kernel stack is 4k or 8k.

• Otherwise similar to user stack exploits: overwrite saved return address with buffer address.

• How do we know where to jump to? And how do we know the location of the saved return address?
Stack overflows

- Typically, kernel stack is 4k or 8k.
- Otherwise similar to user stack exploits: overwrite saved return address with buffer address.
- How do we know where to jump to? And how do we know the location of the saved return address?

```
... thread_info
... shellcode
... ret
... nop
... nop
```

DEEPSEC 2007
Stack overflows

- Typically, kernel stack is 4k or 8k.
- Otherwise similar to user stack exploits: overwrite saved return address with buffer address.
- How do we know where to jump to? And how do we know the location of the saved return address?

```c
&(jmp *%esp)

shellcode

ret
```
Stack overflows

• Typically, kernel stack is 4k or 8k.

• Otherwise similar to user stack exploits: overwrite saved return address with buffer address.

• How do we know where to jump to? And how do we know the location of the saved return address?
Stack overflows

• Typically, kernel stack is 4k or 8k.

• Otherwise similar to user stack exploits: overwrite saved return address with buffer address.

• How do we know where to jump to? And how do we know the location of the saved return address?
Stack overflows

• Typically, kernel stack is 4k or 8k.
• Otherwise similar to user stack exploits: overwrite saved return address with buffer address.
• How do we know where to jump to? And how do we know the location of the saved return address?

```
...  
jmp -N
&(jmp **%esp)
...

...  
shellcode
...
nop
nop

thread_info
```

← ret
Stack overflows

- Typically, kernel stack is 4k or 8k.
- Otherwise similar to user stack exploits: overwrite saved return address with buffer address.
- How do we know where to jump to? And how do we know the location of the saved return address?
Stack overflows

- Typically, kernel stack is 4k or 8k.
- Otherwise similar to user stack exploits: overwrite saved return address with buffer address.
- How do we know where to jump to? And how do we know the location of the saved return address?

```
...  
jmp -N  
&(jmp *%esp)  
&(ret)  
...  
&(ret)  
...  
shellcode  
...  
nop  
nop  
...  
thread_info  
```

- esp  
- ret
Stack overflows

- Typically, kernel stack is 4k or 8k.
- Otherwise similar to user stack exploits: overwrite saved return address with buffer address.
- How do we know where to jump to? And how do we know the location of the saved return address?
Stack overflows

- Typically, kernel stack is 4k or 8k.
- Otherwise similar to user stack exploits: overwrite saved return address with buffer address.
- How do we know where to jump to? And how do we know the location of the saved return address?

```
...  
jmp -N  
&(jmp *%esp)  
&(ret)  
...  
&(ret)  
...  
shellcode  
...  
nop  
nop  
...  
thread_info
```
Inside ring0, now what?
Inside ring0, now what?
Inside ring0, now what?
Inside ring0, now what?
Inside ring0, now what?
Inside ring0, now what?
Inside ring0, now what?
Inside ring0, now what?
Return to ring3!

attacker

r3 shellcode

r0 exploit

target
Metasploit’s approach

• Migration (not implemented yet)
• Stager
• Recovery
• Stage (regular ring3 payloads)
The goal of this step, is to transition to a state where the ring0 payload can be executed in a safe manner.

On Windows it may be necessary to adjust the current IRQL. On Linux, it may be necessary to cleanly get out from an interrupt or softirq.

May coincide with the stager component.
Stager

- Copy the ring0 or ring3 to a suitable location
  - We may only be able to access currently loaded pages
  - Space between kernel stack and thread_info
  - Unused entries in the IDT

- Install hook that will execute the payload in the desired context
  - Interrupt handlers
  - System call handlers (how do we find the system call table?)
Recovery

• If the system crashes after the stager has finished, we haven’t accomplished anything

• We need to recover from the exploit and leave the system in a safe state

• Recovery depends on the situation:
  • Restore registers (but we smashed the stack…)
  • Enable interrupts or preemption
  • Release spinlocks
• Ideally, the stage is simply a ring3 shellcode

• Depending on the migration / stager we may have a two-level stage
  
  • Copy ring3 payload to user-space (in context of a user-mode process)
  • Adjust process privileges ;-)  
  • Set process saved instruction pointer or some function pointer to payload
  • We hooked the sys_execve system call and replaced the command to be executed
conclusion
Conclusion

• Fuzzing 802.11 live on the air is a cumbersome and time-consuming process due to the limitations and requirements of the wireless medium.

• Moving the fuzzer and the target into an emulated environment dramatically speeds up and simplifies the process!

• Every kernel vulnerability is a story of its own, but some generalizations are still possible.
**Fabrice Bellard.** “QEMU, a Fast and Portable Dynamic Translator”
USENIX 2005 Annual Technical Conference

**sgrakkyu & twiz.** “Attacking the Core: Kernel Attacking Notes”
Phrack 0x0c, 0x40, #0x06

**QEMU**
http://www.qemu.org
kudos & respect

Christopher Kruegel
Engin Kirda
http://www.seclab.tuwien.ac.at

Bernhard Müller
http://www.sec-consult.com
vielen dank