Wi-Fi Advanced Fuzzing

Laurent BUTTI – France Télécom / Orange Division R&D

firstname dot lastname at orange-ftgroup dot com

orange™

research & development

0
Forewords

Who Am I?

- Network security expert at R&D labs
  - Working for France Telecom – Orange (a major telco)
- Speaker at security-focused conferences
  - ToorCon, ShmooCon, FIRST, BlackHat US, hack.lu …
- Wi-Fi security centric ;-)
  - “Wi-Fi Security: What’s Next” – ToorCon 2003
  - “Design and Implementation of a Wireless IDS” – ToorCon 2004 and ShmooCon 2005
  - “Wi-Fi Trickery, or How To Secure (?), Break (??) and Have Fun With Wi-Fi” – ShmooCon 2006
  - “Wi-Fi Advanced Stealth” – BlackHat US 2006 and Hack.LU 2006
    - Some words also on 802.11 fuzzing…

Released (Some) Tools

- Last year we released new tools and techniques
  - Raw Fake AP: an enhanced fake AP tool using RAW injection for increased effectiveness
  - Raw Glue AP: a virtual AP catching every client in a virtual quarantine area
  - Raw Covert: a 802.11 tricky covert channel using valid ACK frames
  - Advanced Stealth Patches: madwifi patches to acheive stealth at low cost
    - Tricks to hide yourself from scanners and wireless IDSes
- All this stuff is available at
  - http://rfakeap.tuxfamily.org

Agenda

- 802.11 overview
- What is fuzzing?
- Design and implementation of a 802.11 fuzzer
- (Some) discovered vulnerabilities
- A real-world example: the madwifi vulnerability
- Final words and demonstrations

Overview

- A new vulnerability will be disclosed
- The “fuzzing tool” will not be released today
- But some 802.11 fuzzing scripts will be described
- Will demystify 802.11 driver vulnerabilities
- Talk focused on vulnerability discovery not exploitation
- If Murphy’s law is wrong, some (working) demonstrations ;-)

1 Introduction

What We Were Aware of…

- Wi-Fi weakens entreprise's perimetric security
  - Weak Wi-Fi network infrastructures (open, WEP, misconfigured WPA)
  - Rogue or misconfigured access points (open access points)
- But also weakens client's security
  - Rogue access points in public zones (conferences, hot spots…)
  - Fake access points attacking (automagically) clients [KARMA]
  - Traffic injection within clients' communications [AIRPWN, WIFITAP]
- Unfortunately all these issues are hardly detectable
  - Without specific tools (Wireless IDS…)

What We Guessed…

- Implementation bugs in 802.11 drivers
  - Developed in C/C++
  - Numerous chipsets ⇨ Numerous developers ⇨ Heterogeneous implementations regarding security
    - Equipment manufacturers (not chipsets') ⇨ Obsolete driver packages
- Promising implementation bugs!
  - Potential arbitrary ring0 (kernel) code execution
    - Bypassing all classic security mechanisms: AV, PFW, HIPS…
  - Remotely triggerable within the victim's radio coverage
    - Not necessarly been associated to a rogue access point!
- Quite cool, no?!? ☺

What Happened…

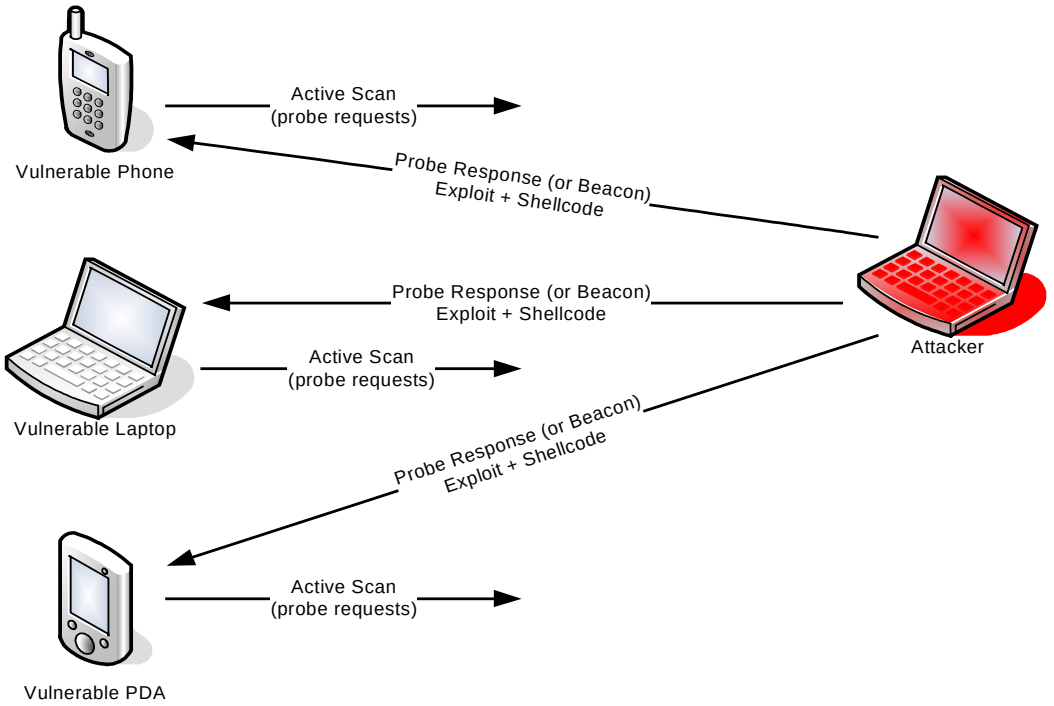
- First public announcement at BlackHat US 2006
  - Johnny Cache and David Maynor presentation [DEVICEDRIVERS]
- Month of Kernel Bugs on November, 2006 [MOKB]
  - Apple Airport 802.11 Probe Response Kernel Memory Corruption (OS X)
  - Broadcom Wireless Driver Probe Response SSID Overflow (Windows)
  - D-Link DWL-G132 Wireless Driver Beacon Rates Overflow (Windows)
  - NetGear WG111v2 Wireless Driver Long Beacon Overflow (Windows)
  - NetGear MA521 Wireless Driver Long Rates Overflow (Windows) (*)
  - NetGear WG311v1 Wireless Driver Long SSID Overflow (Windows) (*)
  - Apple Airport Extreme Beacon Frame Denial of Service (OS X)
- But also under Linux
  - Madwifi stack-based overflow (*)
    - Potentially all recent unpatched Linux distributions running on an Atheros chipset

(*) found by our fuzzer

Potential Targets?

- Nowadays Wi-Fi technologies are ubiquitous!
  - All recent laptops
  - Most entreprises are equipped with Wi-Fi devices
  - More and more home boxes (DSL gateways…)
  - More and more cellular phones (VoIPoWLAN)
  - Video gaming consoles, digital cameras, printers…
- But also, protection / analyser mechanisms may be vulnerable
  - e.g. wireless IDS/IPS, sniffers (tcpdump)…
- So many (potentially) vulnerable Wi-Fi implementations! ☺

802.11 Station Attack Overview



- 802.11 exploits a.k.a. 0wn3d by a 802.11 frame! ;-)

Observations

- Device drivers are potentially less audited than mainline kernels (Windows, Linux)
- If so, 802.11 drivers may be remotely exploitable to gain ring0 privileges
  - Within radio coverage of the victim
- Most chipset manufacturers were hit by implementation bugs
  - Atheros, Intel, Broadcom, Realtek, Orinoco…
- Preventing exploitation means
  - Updating its driver (if patched driver is available!)
  - Switching off the wireless switch (or removing the wireless NIC)

1st Step: Finding These Vulnerabilities!

- Closed source drivers
  - Black box testing
  - Reverse engineering
- Open source drivers
  - Black / White box testing
  - Source code auditing
- Reverse engineering drivers is time consuming
  - Especially when you haven't any clue…
- Source code auditing is only possible if source code is available! ☹
- Black box testing may be useful in both cases…

2

802.11 Fuzzing?

Fuzzing? (1/2)

- Really hard to define…
  - Security community / industry loves this kind of hyped / buzzed words! ;-)
- Some definitions
  - Fuzz Testing or Fuzzing is a Black Box software testing technique, which basically consists in finding implementation bugs using malformed or semi malformed data injection in a automated fashion. [OWASP]
  - Fuzz testing or fuzzing is a software testing technique. The basic idea is to attach the inputs of a program to a source of random data ("fuzz"). If the program fails (for example, by crashing, or by failing built-in code assertions), then there are defects to correct. [WIKIPEDIA]
- Common part
  - Software testing technique that consists in finding implementation bugs
    - 1st definition: with malformed or semi malformed data injection
    - 2nd definition: with random data

Fuzzing? (2/2)

- Fuzzing is by far one of the best price / earning ratio ;-)
  - Reverse engineering load of drivers is costly and boring
  - Implementing a basic fuzzer may be low cost
  - Discovered implementation bugs will thus be the most obvious ones
- But fuzzing will (probably) not help you finding ‘complex’ bugs
  - Simply because all testing possibilities cannot be performed due to
    - Lack of time versus all test possibilities
    - Protocol specificities (states)

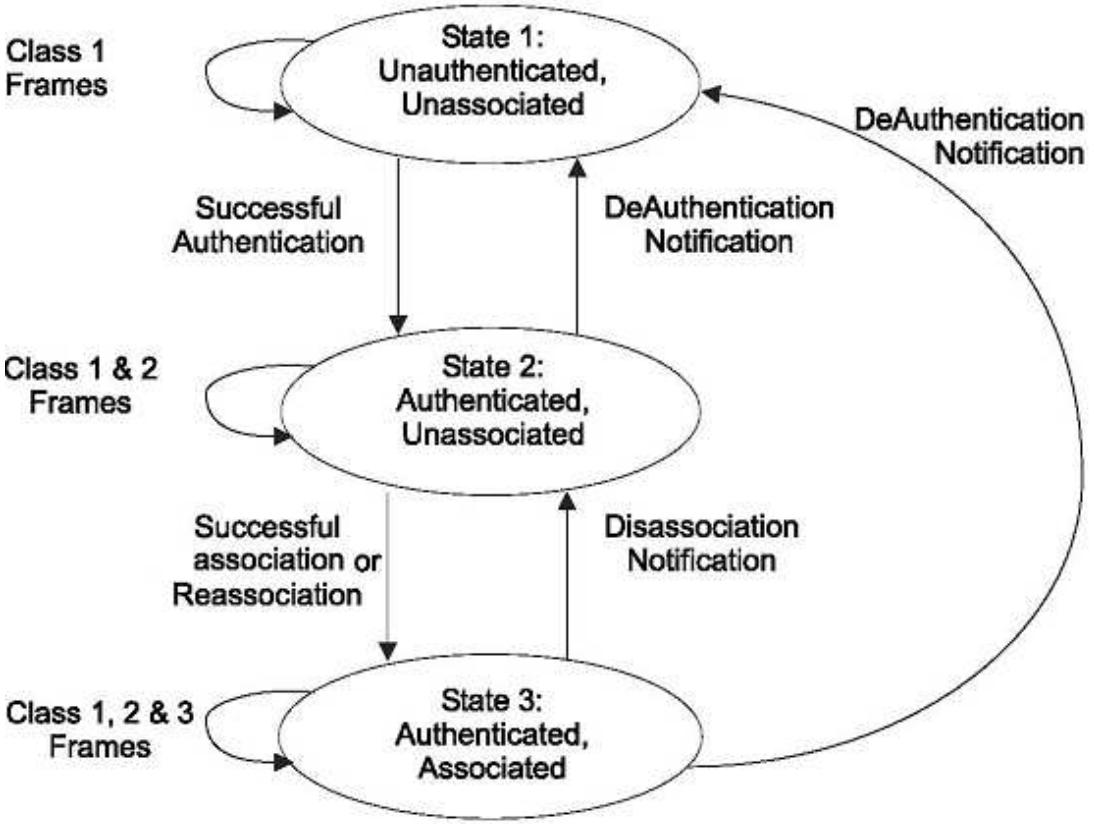
Some Fuzzing Successes

- Month of Browser Bugs and Month of Kernel Bugs
  - Most vulnerabilities discovered thanks to fuzzing techniques
- Take a look at LMH's **`fsfuzzer`** [FSFUZZER]
  - Really basic but _so_ effective! ☺
- Some open source fuzzers
  - SPIKE (Immunity): multi-purpose fuzzer [SPIKE]
  - PROTOS suite (Oulu University): SIP, SNMP… [PROTOS]
- A extensive list of fuzzers is available at:
  - http://www.infosecinstitute.com/blog/2005/12/fuzzers-ultimate-list.html

802.11 Fuzzing? (1/3)

- 802.11 legacy standard is somewhat complex
  - Several frame types (management, data, control)
  - Lot of signalling
    - Rates, channel, network name, cryptographic capabilities, proprietary capabilities…
  - All this stuff must be parsed by the firmware/driver!
- 802.11 extensions are more and more complex!
  - 802.11i for security, 802.11e for QoS…
  - 802.11w, 802.11r, 802.11k…
- Complexity++ ⇨ Code++ ⇨ Bugs++

802.11 Fuzzing? (2/3)



- Every 802.11 state is fuzzable
  - State 1: initial start, unauthenticated, unassociated
  - State 2: authenticated, unassociated
  - State 3: authenticated, associated

802.11 Fuzzing? (3/3)

- Client and access point must be synchronized
  - Driver and firmware filter frames regarding their current state
    - e.g. no data packet accepted whenever in state 1 (refer to 802.11 standard)
- Strong constraints
  - To fuzz state 2 to state 3 changes, the client (or access point) must be in state 2
  - When simulating changing states, ACK frames are a big issue to deal with
  - Only state 1 fuzzing is easy thanks to RAW wireless injection
    - i.e. without operating in driver mode

802.11 Overview 101

- 802.11 chipsets generally provides several modes of operation
  - Monitor: listen to 802.11 layer
  - Master: act as an access point
  - AdHoc: act as an AdHoc station
  - Managed: act as a station
- 802.11 state machine is defined in the IEEE 802.11-1999 std.
- Discovering access points (scanning process)
  - Active scanning: send probe requests and listen to probe responses back, and do channel hopping
  - Passive scanning: listen to beacons and do channel hopping
  - Note: drivers may be listening to both beacons and probe responses

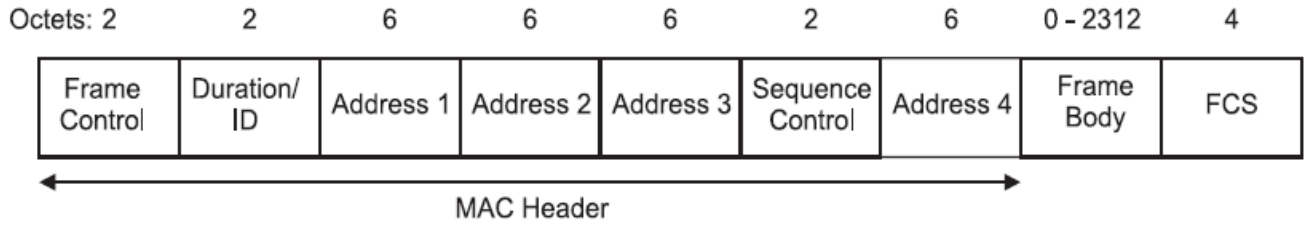
802.11 Overview 101

- 802.11 standard defines 3 class of frames (Chap. 5.5)

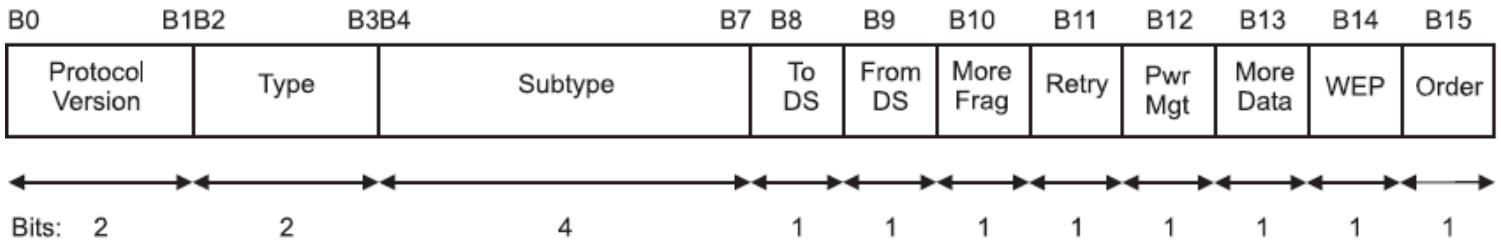
- Management frames regarding the current state
  - Class 1 – permitted from within states 1, 2, and 3
    - Probe Request / Response, Beacon, Authentication Request / Response
    - Deauthentication
  - Class 2 – if & only if authenticated; allowed from within states 2 and 3 only
    - (Re)association Request / Response
    - Deassociation
  - Class 3 – if & only if associated; allowed only from within State 3
    - Deauthentication

802.11 Overview 101

- MAC frame format

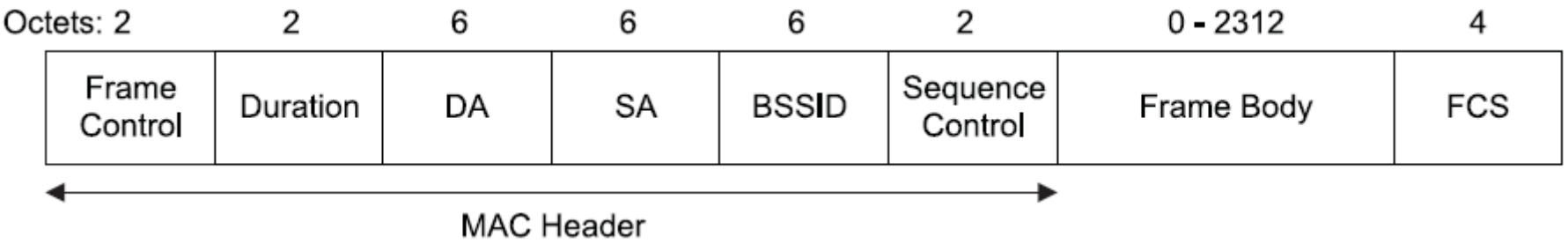


- Frame Control defines upper layer (frame body)



802.11 Overview 101

- Management frame

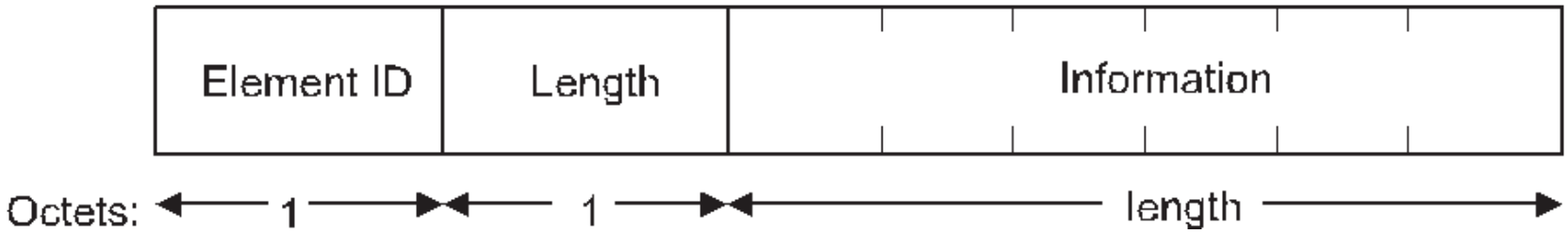


802.11 Overview 101

- Beacon / Probe Response format

Order	Information	Notes
1	Timestamp	
2	Beacon interval	
3	Capability information	
4	SSID	
5	Supported rates	
6	FH Parameter Set	The FH Parameter Set information element is present within Beacon frames generated by STAs using frequency-hopping PHYs.
7	DS Parameter Set	The DS Parameter Set information element is present within Beacon frames generated by STAs using direct sequence PHYs.
8	CF Parameter Set	The CF Parameter Set information element is only present within Beacon frames generated by APs supporting a PCF.
9	IBSS Parameter Set	The IBSS Parameter Set information element is only present within Beacon frames generated by STAs in an IBSS.
10	TIM	The TIM information element is only present within Beacon frames generated by APs.

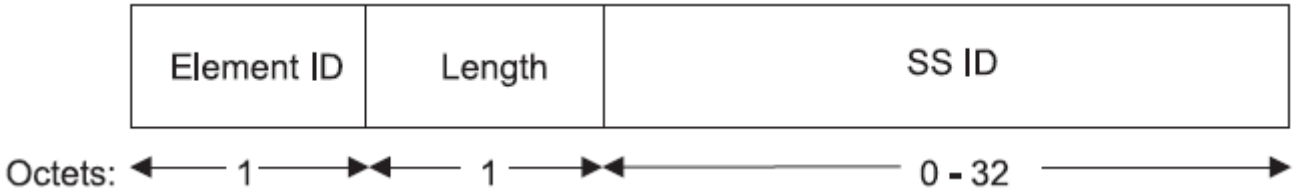
802.11 Overview 101

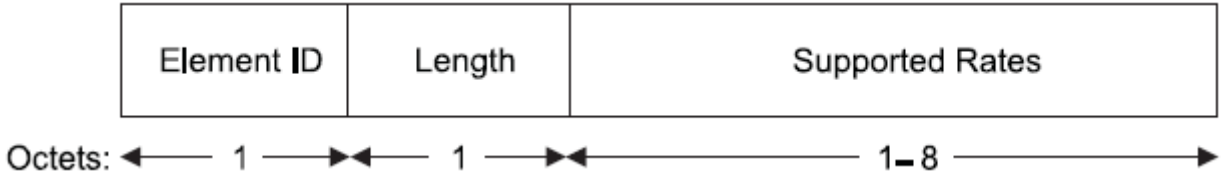


Information element	Element ID
SSID	0
Supported rates	1
FH Parameter Set	2
DS Parameter Set	3
CF Parameter Set	4
TIM	5
IBSS Parameter Set	6
Reserved	7–15
Challenge text	16
Reserved for challenge text extension	17–31
Reserved	32–255

802.11 Overview 101

- Some Information Elements





3

Design and Implementation of a 802.11 Fuzzer

Design of a 802.11 Fuzzer

- Frame injection technique: monitor or master mode?
  - Fuzzing state 1: monitor mode is a good option
  - Fuzzing states 2 and 3: master mode may be the best option
    - Because state changes are managed by the driver, you don't have to emulate this by your userland fuzzer
  - Impacts on development choices
    - Monitor mode allows user-land software
    - Master mode should require driver-land tweaks
  - Depends if your firmware/driver ACKs while in monitor mode…
- Our first approach was to implement state 1 fuzzing

Design of a 802.11 Fuzzer

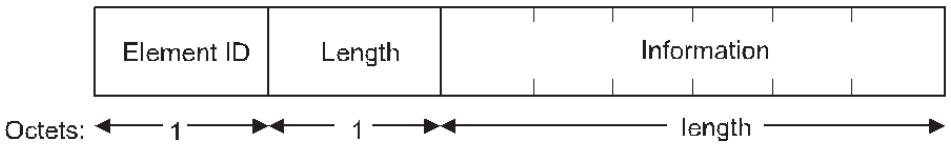
- Taking advantage of active scanning process
  - Precalculate a set of Information Elements to send back
  - Grep for a probe request with Null SSID to the broadcast address
  - Send back appropriate probe response with tested Information Element
- The goal was to optimize testing time
  - Theorically there is no waste of time: 1 request / 1 response
- But we identified a major drawback with this technique
  - You MUST answer very fast as the client device performs channel hopping
    - That is hardly feasible with Python and even harder with Scapy
  - You CANNOT be sure that the frame was analyzed or not by the driver
    - That is a shame because it may induce false negatives ☹

Design of a 802.11 Fuzzer

- But also, as stated in Uninformed Journal #6 [UNINFORMED#6]
  - Some drivers accept beacons ONLY if there are probe responses also!
    - A serious headache!
- The workaround was then to flood the radio with both
  - Beacons to the broadcast address
  - Probe responses to the unicast address of the victim
- Our tool implements different testing strategies (latest the best!)
  - Probe responses triggered by a probe request
  - Probe responses OR beacons during a certain duration
  - Probe responses AND beacons during a certain duration

Design of a 802.11 Fuzzer

- A (good) candidate for 802.11 fuzzing: the Information Element
  - Type / Length / Value
  - Type is the Element ID (1 byte)
  - Length is the total length of the Value payload (1 byte)
  - Value is the payload of the Information Element (0-255 bytes)




```
⊞ IEEE 802.11
⊟ IEEE 802.11 wireless LAN management frame
  ⊞ Fixed parameters (12 bytes)
  ⊟ Tagged parameters (24 bytes)
    ⊟ SSID parameter set: "linksys"
        Tag Number: 0 (SSID parameter set)
        Tag length: 7
        Tag interpretation: linksys
    ⊟ Supported Rates: 1,0(B) 2,0(B) 5,5(B) 11,0(B)
        Tag Number: 1 (Supported Rates)
        Tag length: 4
        Tag interpretation: Supported rates: 1,0(B) 2,0(B) 5,5(B) 11,0(B)  [Mbit/sec]
```

- Some IEs have a fixed or maximum length
  - Possible buffer overflows if not properly checked
  - Static buffer to fit all the payload of the information element
  - Take the length within 802.11 frame
  - If this length is above the static buffer size then it may overflow

Design of a 802.11 Fuzzer

- We defined a list of popular information elements
  - IE 0 : SSID : minimum size of 0 byte, maximum size of 32 bytes
  - IE 3 : Channel : fixed size of 1 byte
  - etc…
- In order to make some fast boundary tests
  - For IE 0, just test for {0, 1, MIN-1, MIN, MIN+1, MAX-1, MAX, MAX+1, 254, 255} length
  - For IE 3, just test for {0, 1, FIXED-1, FIXED, FIXED+1, 254, 255}
  - And so on…
- Goal was to limit test space in order to optimize testing time!
  - Doing it linearly or randomly may success but is quite time consuming!

Design of a 802.11 Fuzzer

- Some information elements are complex
  - WPA, RSN [Security]
  - WMM [Quality of Service]
  - WPS [Wireless Provisioning Services]
  - Proprietary IEs (Atheros, Cisco,…)
- Randomly testing TLVs is interesting but far from effective
  - Parsers generally check what is carried within the information element
    - Thanks to OUIs
  - Functions that understand the underlying protocol are necessary
    - Your fuzzer should be WPA-aware in order to test the WPA capabilities parser
    - etc…
- Testing different code paths is the goal of an efficient fuzzer

Design of a 802.11 Fuzzer

- WPA Information Element example
  - WPA IE (1 byte) ['\xDD']
  - WPA OUI (3 bytes) is mandatory at the beginning of the IE payload
  - WPA TYPE (1 byte) + WPA VERSION (2 bytes)
  - WPA multicast cipher (4 bytes)
  - Number of unicast ciphers (2 bytes: m value)
  - WPA list of unicast ciphers (4*m bytes)
  - Number of authentication suites (2 bytes: n value)
  - WPA list of authentication suites (4 * n bytes)

- Seems to be interesting for possible overflows
  - The WPA-parser must implement numerous checks
    - Theoretical length versus packet length

Design of a 802.11 Fuzzer

- To be effective the fuzzer must be smarter than random
  - Set the beginning of the frame in order to be accepted by the WPA parser
    - WPA IE + WPA OUI + WPA TYPE + WPA VERSION
  - Then you may have a lot of options
    - Vary ‘m’ and ‘n’ values to check for overflows
    - Truncate these frames
    - Fill it with unrelevant values
- So many possibilities but requires more work and testing time…
  - Testing will never be fully exhaustive but should be sufficient to trigger most obvious bugs
- But proprietary IEs are hard to analyze
  - Lack of documentation…

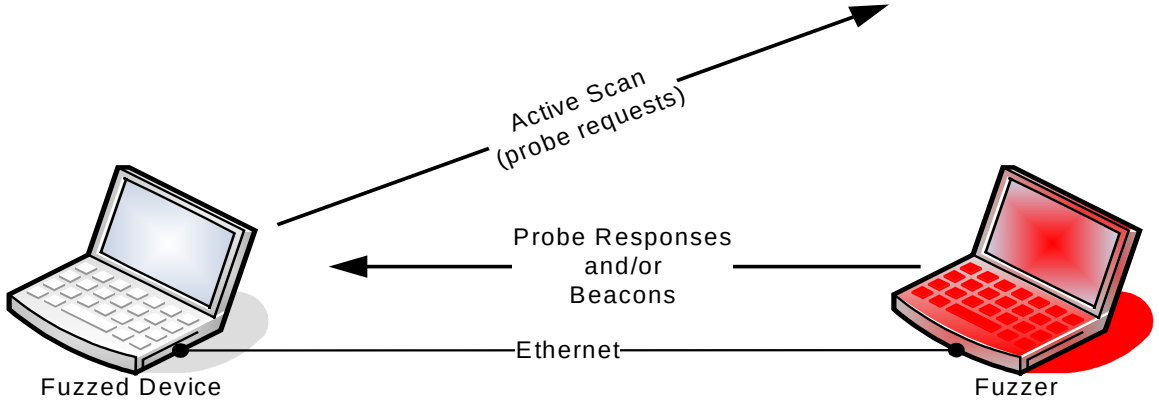
Implementation of a 802.11 Fuzzer

- Re-use existing fuzzing frameworks/software?
  - None of them perfectly fitted our needs
- Re-use existing frame injection frameworks/software?
  - Scapy seems to be the best option [SCAPY]
    - Our first option but probe response mode wasn't effective with scapy (too slow)
- We developed our own tool
  - Python for its flexibility and development speed comparing to C/C++ when formatting 802.11 frames
  - But our testing cases may be used thanks to scapy with "flooding" mode

Implemented Features

- RAW injection of any arbitrary 802.11 frame (monitor mode)
- Smart combination of IEs for improved testing range and reduced testing time
- Specific tests like truncated frames, empty frames…
- Specific testers for
  - {WPA, RSN, WSP} IEs
  - Some proprietary IEs

Architecture Overview



- Ethernet connectivity enables us to detect whenever a bug is triggered (keepalive)

Automated Bug Detection

- On Windows, critical bugs will trigger a BSOD
  - Have a script running on the fuzzing station that pings the fuzzed station and send a SIGINT whenever the victim does not respond
    - The fuzzer should then display the last test that triggered the bug
- On Linux, bugs will trigger a dump via kernel logs (syslog)
  - Have a script running on the fuzzed station that greps for {oops|unable to handle|assert|panic} in kernel messages
- A malfunction may leave the wireless device non functional
  - Have a script listening to station's probe requests and send a SIGINT whenever there is no more probe requests
    - Works only if active scanning
- Of course, works only on critical bugs!

Force Active Scanning

- Cannot state if 802.11 device is listening or not to beacons, forcing it to scan actively for access points will do the job!
- Force the wireless interface to scan for 802.11 access points
  - Under Windows: using Netstumbler
  - Under Linux: using `iwlist` as root (SIOCSIWSCAN and SIOCGIWSCAN)
- Check if a newly created frame is analyzed by the driver
  - In Netstumbler list of access points
  - In `iwlist` list of access points

Modus Operandi (Windows Drivers)

- Set up network connectivity on fuzzing and fuzzed computers
- Set up the fuzzing 802.11 device (monitor mode)
- Set up the fuzzed 802.11 device (scanning mode)
- Set up the script to catch BSODs
- Set up the fuzzer command line
- Launch the fuzzing
- Verify that fuzzing process is OK
- Wait and see ☺

- If the fuzzed device is vulnerable to one particular test
  - SIGINT will enable us to have the test that trigger the bug!

Passing Successive Tests

- Thanks to command-line configuration (or anything else)
  - Precalculate the beacon / probe response 802.11 header
    - Can be static as no checks on sequence numbers nor BSS Timestamps are performed by the drivers
    - Our preliminary fuzzer implemented consistent sequence numbers and BSS Timestamps to bypass possible MAC spoofing detection / prevention
  - Precalculate a set of tests
- Then inject $test_n$ and increment n regarding the duration time
  - Overall testing time is limited
- Show progress to the user
  - May kill the process and the last current test will be displayed

4

Discovered Vulnerabilities

State of The Art: 802.11 Driver/Parser Vulnerabilities

CVE-2007-1218 (PARSER)	Off-by-one buffer overflow in the parse_elements function in the 802.11 printer code (print-802_11.c) for tcpdump 3.9.5 and earlier allows remote attackers to cause a denial of service (crash) via a crafted 802.11 frame. NOTE: this was originally referred to as heap-based, but it might be stack-based.
CVE-2007-0933 (DRIVER/WIN)	Will be released today
CVE-2007-0686 (DRIVER/WIN)	The Intel 2200BG 802.11 Wireless Mini-PCI driver 9.0.3.9 (w29n51.sys) allows remote attackers to cause a denial of service (system crash) via crafted disassociation packets, which triggers memory corruption of "internal kernel structures," a different vulnerability than CVE-2006-6651. NOTE: this issue might overlap CVE-2006-3992.
CVE-2007-0457 (PARSER)	Unspecified vulnerability in the IEEE 802.11 dissector in Wireshark (formerly Ethereal) 0.10.14 through 0.99.4 allows remote attackers to cause a denial of service (application crash) via unspecified vectors.
CVE-2006-6651 (DRIVER/WIN)	Race condition in W29N51.SYS in the Intel 2200BG wireless driver 9.0.3.9 allows remote attackers to cause memory corruption and execute arbitrary code via a series of crafted beacon frames. NOTE: some details are obtained solely from third party information.
CVE-2006-6332 (DRIVER/LIN)	Stack-based buffer overflow in net80211/ieee80211_wireless.c in MadWifi before 0.9.2.1 allows remote attackers to execute arbitrary code via unspecified vectors, related to the encode_ie and giwscan_cb functions.
CVE-2006-6125 (DRIVER/WIN)	Heap-based buffer overflow in the wireless driver (WG311ND5.SYS) 2.3.1.10 for NetGear WG311v1 wireless adapter allows remote attackers to execute arbitrary code via an 802.11 management frame with a long SSID.
CVE-2006-6059 (DRIVER/WIN)	Buffer overflow in MA521nd5.SYS driver 5.148.724.2003 for NetGear MA521 PCMCIA adapter allows remote attackers to execute arbitrary code via (1) beacon or (2) probe 802.11 frame responses with an long supported rates information element. NOTE: this issue was reported as a "memory corruption" error, but the associated exploit code suggests that it is a buffer overflow.
CVE-2006-6055 (DRIVER/WIN)	Stack-based buffer overflow in A5AGU.SYS 1.0.1.41 for the D-Link DWL-G132 wireless adapter allows remote attackers to execute arbitrary code via a 802.11 beacon request with a long Rates information element (IE).
CVE-2006-5972 (DRIVER/WIN)	Stack-based buffer overflow in WG111v2.SYS in NetGear WG111v2 wireless adapter (USB) allows remote attackers to execute arbitrary code via a long 802.11 beacon request.

State of The Art: 802.11 Driver/Parser Vulnerabilities

CVE-2006-5882 (DRIVER/WIN)	Stack-based buffer overflow in the Broadcom BCMWL5.SYS wireless device driver 3.50.21.10, as used in Cisco Linksys WPC300N Wireless-N Notebook Adapter before 4.100.15.5 and other products, allows remote attackers to execute arbitrary code via an 802.11 response frame containing a long SSID field.
CVE-2006-5710 (DRIVER/OSX)	The Airport driver for certain Orinoco based Airport cards in Darwin kernel 8.8.0 in Apple Mac OS X 10.4.8, and possibly other versions, allows remote attackers to execute arbitrary code via an 802.11 probe response frame without any valid information element (IE) fields after the header, which triggers a heap-based buffer overflow.
CVE-2006-3992 (DRIVER/WIN)	Unspecified vulnerability in the Centrino (1) w22n50.sys, (2) w22n51.sys, (3) w29n50.sys, and (4) w29n51.sys Microsoft Windows drivers for Intel 2200BG and 2915ABG PRO/Wireless Network Connection before 10.5 with driver 9.0.4.16 allows remote attackers to execute arbitrary code via certain frames that trigger memory corruption.
CVE-2006-3509 (DRIVER/OSX)	Integer overflow in the API for the AirPort wireless driver on Apple Mac OS X 10.4.7 might allow physically proximate attackers to cause a denial of service (crash) or execute arbitrary code in third-party wireless software that uses the API via crafted frames.
CVE-2006-3508 (DRIVER/OSX)	Heap-based buffer overflow in the AirPort wireless driver on Apple Mac OS X 10.4.7 allows physically proximate attackers to cause a denial of service (crash), gain privileges, and execute arbitrary code via a crafted frame that is not properly handled during scan cache updates.
CVE-2006-3507 (DRIVER/OSX)	Multiple stack-based buffer overflows in the AirPort wireless driver on Apple Mac OS X 10.3.9 and 10.4.7 allow physically proximate attackers to execute arbitrary code by injecting crafted frames into a wireless network.
CVE-2006-1385 (PARSER)	Stack-based buffer overflow in the parseTaggedData function in WavePacket.mm in KisMAC R54 through R73p allows remote attackers to execute arbitrary code via multiple SSIDs in a Cisco vendor tag in a 802.11 management frame.
CVE-2006-0226 (DRIVER/BSD)	Integer overflow in IEEE 802.11 network subsystem (ieee80211_ioctl.c) in FreeBSD before 6.0-STABLE, while scanning for wireless networks, allows remote attackers to execute arbitrary code by broadcasting crafted (1) beacon or (2) probe response frames.

State of The Art: 802.11 Driver/Parser Vulnerabilities

- Tentative overview so some vulnerabilities may be missing…

- 18 CVE entries
  - 15 are driver related
    - 9 Windows
    - 4 OS X
    - 1 Linux
    - 1 FreeBSD (much more 802.11 subsystem than driver but it is kernel-land)
  - 3 are sniffer / parser related
    - Ethereal / Wireshark and tcpdump
    - KisMAC

- First entry was the FreeBSD integer overflow (beginning of 2006)

State of The Art: 802.11 Driver/Parser Vulnerabilities

- Among 14 _different_ driver related vulnerabilities
  - Long SSID (x3), Long Supported Rates (x2), Long TIM (x1)
    - Easy to discover thanks to fuzzing
  - Set of long IEs
    - Easy to discover thanks to fuzzing
  - No valid IE
    - Easy to discover thanks to fuzzing
  - IE WPA/RSN/WMM (madwifi and FreeBSD vulnerabilities)
    - Needs a generic OUI fuzzer
  - (Flood of) disassociation packets
    - Hard to discover (needs to be in state 3)
  - 3 are unspecified

802.11 Driver Vulnerabilities Discovered Thanks to Our Fuzzer

- NetGear MA521 Wireless Driver Long Rates Overflow
  - Overflowing Rates Information Element
    - This field has generally a maximum length of 8 bytes (implementation dependent)
- NetGear WG311v1 Wireless Driver Long SSID Overflow
  - Overflowing SSID Information Element
    - This field has a maximum length of 32 bytes
- D-Link DWL-G650+ (A1) Wireless Driver Long TIM Overflow
  - Overflowing TIM Information Element
- Madwifi Driver Remote Buffer Overflow Vulnerability
  - Overflowing WPA/RSN/WMM/ATH Information Element
  - Triggered when SIOCGIWSCAN
    - e.g. thanks to **`iwlist`** or **`iwlib.h`**

Firmware Bugs

- With some chipsets, state 1 is managed by the firmware
  - Thus discovered bugs will be within the firmware
- NULL probe responses may leave the device non functional
  - 802.11b Firmware-Level Attacks [FIRMWAREBUGS]
- Some chipsets
  - Prism54
  - Prism2.5
- Hard to detect them automatically
  - ‘Scheduling firmware restart’ under Linux
  - (Generally) no scan results under Netstumbler and thus requires a down&up of the wireless interface under Windows

Madwifi Example

- This bug was discovered thanks to a specific WPA tester
  - Required to have a valid WPA (OUI + TYPE + VERSION) in the IE payload
- Vulnerable code is located in

  `net80211/ieee80211_wireless.c`
- Static buffer definition in `giwscan_cb()`

```
#if WIRELESS_EXT > 14
  char buf[64 * 2 + 30];
#endif
```

- Requires kernel > 2.4.20, 2.5.7 [WIRELESSTOOLS]

Madwifi Example

```
#ifdef IWEVGENIE
        memset(&iwe, 0, sizeof(iwe));
        memcpy(buf, se->se_wpa_ie, se->se_wpa_ie[1] + 2);
        iwe.cmd = IWEVGENIE;
        iwe.u.data.length = se->se_wpa_ie[1] + 2;
#else
        static const char wpa_leader[ ] = "wpa_ie=";
        memset(&iwe, 0, sizeof(iwe));
        iwe.cmd = IWEVCUSTOM;
        iwe.u.data.length = encode_ie(buf, sizeof(buf),
                se->se_wpa_ie, se->se_wpa_ie[1] + 2,
                wpa_leader, sizeof(wpa_leader) - 1);
#endif
```

Buffer overflow

encode_ie() vulnerable

- Same code for RSN and WME Information Elements

Madwifi Example

- 1st security bug
  - `memcpy(buf, se->se_wpa_ie, se->se_wpa_ie[1] + 2);`
  - `se->se_wpa_ie[1]` is the IE length in the 802.11 frame
    - Could be 255 thus the copied length may be 257 bytes
    - Overflowing the static buffer!

Data controlled by the attacker

- 2nd security bug (in encode_ie())
  - `for (i = 0; i < ielen && bufsize > 2; i++) p += sprintf(p, "%02x", ie[i]);`
  - `p` is a pointer to static buffer `buf`
  - `ielen` is the IE length in the 802.11 frame
    - Could be 257
    - Overflowing the static buffer!

Madwifi Example

- These bugs were triggered thanks to a SIOCGIWSCAN
  - **`iwlist`** gets scanning results from the driver
  - Vulnerable code is executed _only_ when SIOCGIWSCAN
  - Vulnerability is triggered _only_ if malformed 802.11 frame is parsed by the vulnerable code
    - Thanks to a SIOCSIWSCAN (doing a 802.11 scan)
- Whenever you ‘up’ the wireless card, there is a scan
  - Thus if you execute **`iwlist ath0 scanning`** (non root) the driver may parse a malformed 802.11 frame
  - But any other application using wireless-tools API would trigger the bug

Madwifi Example

- It is a stack overflow

```
BUG:
unable to handle kernel paging request at virtual address 45444342
 printing eip:
45444342
*pde = 00000000
Oops: 0000 [#1]
PREEMPT
CPU:    0
EIP:    0060:[<45444342>]    Tainted: P      VLI
EFLAGS: 00210282   (2.6.17.11 #1)
EIP is at 0x45444342
eax: 00000000   ebx: 41414141   ecx: 00000000   edx: f4720bde
esi: 41414141   edi: 41414141   ebp: 41414141   esp: f3f2be24
ds: 007b   es: 007b   ss: 0068
Process iwlist (pid: 3486, threadinfo=f3f2a000 task=f6f8a5b0)
```

Madwifi Example

- Rewrite EIP to a stable address (e.g. that is a JMP ESP)
- Put a JMP BACK to the core of the 802.11 Information Element
- If you use a JMP SHORT (-128 bytes maximum), then you may use a 2nd JMP SHORT to the beginning of the Information Element

- You have about 158 bytes (buffer length) – 6 bytes (WPA header) for the shellcode
  - If buffer is not modified before returning to the calling function

Madwifi Example

- We contacted Madwifi team on December, 5th
- They released a patched package (0.9.2.1) on December, 6th
- We disclosed on DailyDave on December, 7th
  - Madwifi SIOCGIWSCAN vulnerability (CVE-2006-6332)
- We released a local exploit on DailyDave on December, 8th
  - Metasploit module for DoS and triggering the local exploit and the local exploit itself

- We thank the Madwifi team for their responsiveness

Madwifi Example

- To that date, not all Linux distributions packaged the patched madwifi driver
  - SUSE was the first
  - Ubuntu did it recently
- You may be vulnerable if you did not manually patched your madwifi driver!

5
Automated Exploitation

Automated Exploitation

- Loss of Radio CONnectivity is a library for RAW injection independent of the underlying chipset/driver [LORCON]
  - LORCON integration in Metasploit since 3.0
    - ruby-lorcon bindings
- Creating and sending a frame is as easy as
  - frame = ''\x00\x00''
  - wifi.write(frame)
- Examples
  - http://metasploit.com/svn/framework3/trunk/modules/auxiliary/dos/wireless/netgear_ma521_rates.rb
  - http://metasploit.com/svn/framework3/trunk/modules/auxiliary/dos/wireless/netgear_wg311pci.rb

Automated Exploitation

- Have a listener greping for 802.11 frames
  - Fingerprint their 802.11 devices
    - By their MAC address (OUI)
    - By their radio capabilities (rates, proprietary information elements…)
    - By their behaviour (RTS/CTS, duration ID) [DEVICEDRIVERS]
  - Elect an appropriate exploit
  - Exploit it!
- Should be easily automated thanks to Metasploit and some basic scripting…

6

802.11 Fuzzing With (Great) Open Source Tools

Fuzzing With Scapy

- `fuzz()` provides means to randomly generate values for any field you did not supply

- Want to randomly fuzz IEs in beacons?
  - `frame=Dot11(proto=0,FCfield=0,ID=0,addr1=DST,addr2=BSSID, addr3=BSSID,SC=0,addr4=None)/Dot11Beacon(beacon_interval= 100,cap="ESS")/Dot11Elt()`
- Want to randomly fuzz SSIDs in beacons?
  - `frame=Dot11(proto=0,FCfield=0,ID=0,addr1=DST,addr2=BSSID, addr3=BSSID,SC=0,addr4=None)/Dot11Beacon(beacon_interval= 100,cap="ESS")/Dot11Elt(ID=0)`
- Want to randomly fuzz 802.11 packets?
  - `frame=Dot11(addr1=DST,addr2=BSSID,addr3=BSSID,addr4=None)`

- Sent it thanks to: `sendp(fuzz(frame))`

Fuzzing With MetaSploit

- Introduced since LORCON integration
  - http://metasploit.com/svn/framework3/trunk/modules/auxiliary/dos/wireless/fuzz_beacon.rb
  - http://metasploit.com/svn/framework3/trunk/modules/auxiliary/dos/wireless/fuzz_proberesp.rb
- These plugins use random technique for IEs
  - But really effective for most obvious bugs (e.g. not madwifi's)
- Fuzzing 802.11 stacks thanks to command-line ;-)
  - `./msfcli auxiliary/dos/wireless/fuzzproberesp DRIVER=madwifing ADDR_DST=11:22:33:44:55:66 PING_HOST=192.168.1.10 E`

7
Detection and Prevention

Exploits Detection?

- Exploits try to trigger a vulnerability
- Thus most of wireless exploits may be detected thanks to
  - Signature-based Wireless IDS
    - Detecting presets of exploits
  - Anomaly-based Wireless IDS
    - Detecting non standard 802.11 packets (oversized information elements…)
- We will probably see more and more exploit signatures in Wireless IDSs

Exploits Prevention?

- Patch your wireless drivers may prevent them
  - Patch and cross your fingers! ;-)
- Otherwise, turn off the wireless switch!

8 Feedbacks

Drawbacks and Issues

- Using a fuzzer may be harder than coding one
  - Setting up the architecture, drivers and tools
  - A strict process must be followed to avoid false negatives and re-testing
  - Be able to replay the bug whenever the bug is triggered
- Bugs must be easily replayable
  - In order to speed up investigation
- Fuzzing some devices may be difficult to achieve
  - Wi-Fi enabled phones (usually) does not have stumblers
    - In order to force active scanning
- So many reboots… ☹

9
Future Work

Fuzzing Other 802.11 States

- Up to now we have presented the fuzzing of client's scanning (state 1)
  - Probe requests ⇨ Probe response / Beacons
- Authentication procedure may be fuzzed (state 1 ⇨ state 2)
  - Authentication response with shared secret is a TLV thus may be fuzzed
    - Challenge from the access point
- Association procedure may be fuzzed (state 2 ⇨ state 3)
  - Association response with client-parsed IEs

Fuzzing 802.11 Access Points

- Fuzzing state 1 of access points may be easily implemented in our fuzzer
  - Replacing the Probe Response with Probe Request
- But some (new) constraints must be taken into account
  - Access point firmware greps for its configured SSID regarding the received probe requests
  - Testing candidates must be tuned for this…
- Fuzzing other states for access points need also more work
- Ongoing work… more on this soon!

10
Final Words

Conclusions

- Fuzzing is quite interesting whenever low-cost black-box testing is required
- It is designed to discover most obvious bugs
- It may be improved to detect more complex bugs but requires a serious understanding of the tested protocol
- It enabled us to discover several critical bugs
  - Only with a state 1 fuzzer…

Conclusions

- Fuzzing 802.11 is only at its beginning
  - New 802.11 extensions are coming
  - But will require smart fuzzers
- Fuzzing 802.11 access points firmwares is the next step
  - Triggering DoS
- Fuzzing other wireless devices will be attractive
  - Wireless USB
  - WiMAX
  - …

Acknowledgements

- Yoann Guillot, Matthieu Maupetit, Jérôme Razniewski, Raphaël Rigo, Julien Tinnès, Franck Veysset

11 Appendices

References

- [KARMA] – D. Dai Zovi & S. Macaulay, KARMA
- [AIRPWN] – airpwn
- [WIFITAP] – Cédric Blancher, wifitap
- [DEVICEDRIVERS] – Johnny Cache & David Maynor, Device Drivers
- [MOKB] – L.M.H., Month of Kernel Bugs
- [OWASP] – Open Web Application Security Project
- [WIKIPEDIA] – The Free Encyclopedia
- [FSFUZZER] – L.M.H., fsfuzzer
- [SPIKE] – Immunity, SPIKE
- [PROTOS] – Oulu University, Security Testing of Protocol Implementations

References

- [SCAPY] – Philippe Biondi, scapy
- [UNINFORMED#6] – Johnny Cache, HD Moore & skape, Exploiting 802.11 Wireless Driver Vulnerabilities on Windows
- [FIRMWAREBUGS] – Joshua Wright & Mike Kershaw, 802.11b Firmware-Level Attacks
- [WIRELESSTOOLS] – Jean Tourrilhes, Wireless Tools for Linux
- [LORCON] – Joshua Wright & Mike Kershaw, LORCON
- [MSPLOIT] – Metasploit Framework

References (Bibliography)

- Laurent Butti & Franck Veysset – Wi-Fi Security: What's Next
- Laurent Butti & Franck Veysset – Design and Implementation of a Wireless IDS
- Laurent Butti & Franck Veysset – Wi-Fi Trickery, or How To Secure (?), Break (??)…
- Laurent Butti & Franck Veysset – Wi-Fi Advanced Stealth
- Laurent Butti – Raw Fake AP, Raw Glue AP, Raw Covert, Wi-Fi Advanced Stealth Patches, http://rfakeap.tuxfamily.org

References (Discovered Vulnerabilities)

- CVE-2006-6059 – Netgear MA521 Wireless Driver Long Rates Overflow

```
DRIVER_IRQL_NOT_LESS_OR_EQUAL (d1)
An attempt was made to access a pageable (or completely invalid) address
at an interrupt request level (IRQL) that is too high.
This is usuallycaused by drivers using improper addresses.
If kernel debugger is available get stack backtrace.
Arguments:
Arg1: 2c2b2a29, memory referenced
Arg2: 00000002, IRQL
Arg3: 00000000, value 0 = read operation, 1 = write operation
Arg4: aa2cc75a, address which referenced memory
```

References (Discovered Vulnerabilities)

- CVE-2006-6125 – NetgearWG311v1 Wireless Driver Long SSID Overflow

```
BAD_POOL_HEADER (19)
The pool is already corrupt at the time of the current request.
This may or may not be due to the caller.
The internal pool links must be walked to figure out a possible cause of
the problem, and then special pool applied to the suspect tags or the
driver verifier to a suspect driver.
Arguments:
Arg1: 00000020, a pool block header size is corrupt.
Arg2: 81cae7b0, The pool entry we were looking for within the page.
Arg3: 81cae8c8, The next pool entry.
Arg4: 0a23002b, (reserved)
```

References (Discovered Vulnerabilities)

- CVE-2006-6332 – Madwifi Driver giwscan_cb() and encode_ie() Remote Buffer Overflow Vulnerability

```
BUG: unable to handle kernel paging request at virtual address 45444342
 printing eip:
45444342
*pde = 00000000
Oops: 0000 [#1]
PREEMPT
CPU:    0
EIP:    0060:[<45444342>]    Tainted: P      VLI
EFLAGS: 00210282   (2.6.17.11 #1)
EIP is at 0x45444342
eax: 00000000   ebx: 41414141   ecx: 00000000   edx: f4720bde
esi: 41414141   edi: 41414141   ebp: 41414141   esp: f3f2be24
ds: 007b   es: 007b   ss: 0068
Process iwlist (pid: 3486, threadinfo=f3f2a000 task=f6f8a5b0)
```

References (Discovered Vulnerabilities)

- CVE-2007-0993 – D-Link DWL-G650+ Wireless Driver Long TIM Overflow

```
DRIVER_IRQL_NOT_LESS_OR_EQUAL (d1)
An attempt was made to access a pageable (or completely invalid) address
at an interrupt request level (IRQL) that is too high.
This is usually caused by drivers using improper addresses.
If kernel debugger is available get stack backtrace.
Arguments:
Arg1: 00760010, memory referenced
Arg2: 00000002, IRQL
Arg3: 00000000, value 0 = read operation, 1 = write operation
Arg4: aa1028de, address which referenced memory
```