J Comput Virol (2008) 4:25–37
DOI 10.1007/s11416-007-0065-x

SSTIC 2007 BEST ACADEMIC PAPERS

Discovering and exploiting 802.11 wireless driver vulnerabilities

Laurent Butti · **Julien Tinnès**

Received: 5 January 2007 / Revised: 15 July 2007 / Accepted: 4 September 2007 / Published online: 29 September 2007
© Springer-Verlag France 2007

Abstract 802.11 Wireless local area networks are unfortunately notoriously infamous due to their many, critical security flaws. Last year, world-first 802.11 wireless driver vulnerabilities were publicly disclosed, making them a critical and recent threat. In this paper, we expose our research results on 802.11 driver vulnerabilities by focusing on the design and implementation of a fully featured 802.11 fuzzer that enabled us to find several critical implementation bugs that are potentially exploitable by attackers. Lastly, we will detail the successful exploitation of the first 802.11 remote kernel stack overflow under Linux (madwifi driver).

1 Introduction

A 2006 landmark has been the presentation "Device Drivers" [2] by Johnny Cache and David Maynor at Black Hat USA [3]. They detailed the current vulnerability research that focuses more and more on device driver vulnerabilities as the source code of main kernels, operating systems and software is much more secure than years ago: it is much harder to find and exploit a system vulnerability[1] than before. Efforts on secure programming and code auditing besides overflow prevention mechanisms in most kernels are helpful in the prevention of finding and exploit a security vulnerability. That is the reason why finding vulnerabilities in third-party device drivers may be of interest for vulnerability researchers.

During their presentation, Cache and Maynor played a video [4] showing a successful exploitation of a MacBook computer thanks to a wireless vulnerability. This video was impressive for the audience as it was the first time people made the disclosure of a potential exploitable security vulnerability in a wireless driver. But, the vulnerability disclosure thanks to a video was quite disappointing as no further precisions were disclosed by the researchers. Moreover, the video had some incoherence that issued a lot of discussions [5,6]. At the present, it is still hard to understand what really happened even if some public explanations are available [7–9].

The main interest of this presentation was to make the people aware of wireless driver vulnerabilities that are inherently critical as any exploitable security bug will be by the "air". An attacker would be able to perform remotely arbitrary code execution with kernel privileges—also called ring0 [10]. Given that most laptops are today shipped with 802.11 devices and that most people now communicates more with 802.11 technology than ever, consequently the risk is quite high. Of course, classic security mechanisms like personal firewalls, anti-virus and host intrusion prevention systems are completely inefficient as the attacker is able to run code with kernel privileges. Would it be possible to compromise your operating system thanks to wireless driver vulnerabilities?

2 Context

A driver is usually coded in C/C++ languages and classic implementation bugs are possible even in these extremely critical parts of the operating system. One could imagine that drivers should be much more code audited since they are running with kernel privileges; but with third-party drivers, it is far from being than mandatory. Classic programming

L. Butti (✉) · J. Tinnès
France Telecom Orange Labs, Network and Service Security Labs,
38-40 Rue du General Leclerc,
92794 Issy-les-Moulineaux Cedex 9, France
e-mail: laurent.butti@orange-ftgroup.com

[1] In this sentence, we do not refer to web-based vulnerabilities such as PHP, SQL injection and XSS that are so numerous…

errors at the user-land level may be also present in kernel-land applications like wireless drivers.

So, the first step is discovering these implementation bugs that could be exploitable in some particular cases. As a matter of fact, successful (and reliable) exploitation is not sure for any vulnerability, and this must be carefully evaluated (limited stack overflow, hard to exploit off-by-one, address space layer randomization…).

Regarding security, most of the time the inherent complexity of protocols and applications makes security analysis very difficult not to say very hard—due to the fact that implementation of a complex application is error-prone. In the present paper, we focus on the 802.11 standard and its extensions [1] (802.11i for enhanced security, 802.11e for quality of service, and the next ones…) that are quite complex and thus error-prone, when implementing the firmware and drivers in both wireless client and access points. Most of the time, finding vulnerabilities by direct source code auditing and/or reverse engineering can be quite time consuming and requires some advanced skills and tools. In other words: it is extremely costly! That is the reason why using other techniques to find implementation bugs must be evaluated in order to have a better price/earning trade-off.

3 Fuzzing

3.1 Definitions

A number of definitions are available on the Internet, so we decided to extract these two ones hereafter below that seem to be quite relevant to us:

- “Fuzz testing or fuzzing is a software testing technique that provides random data (“fuzz”) to the inputs of a program. If the program fails (for example, by crashing, or by failing built-in code assertions), the defects can be noted” (source: Wikipedia, the free encyclopedia).
- “Fuzz testing or Fuzzing is a Black Box software testing technique, which basically consists in finding implementation bugs using malformed/semi-malformed data injection in an automated fashion” (source: Open Web Application Security Project).

Thus we can define fuzzing as a software testing technique aiming at finding implementation bugs in an automated way. We consider that fuzzing relying only on malformed/semi-malformed data can be quite misleading, i.e. in some situations a valid frame according to a protocol standard (RFC, IEEE) may be invalid with respect to the software (crashing the application). So in this particular case, the malformed/semi-malformed frame is only bad for the application that contains an implementation bug.

Even if a great fuss is made about it since 2 years, fuzzing is not a recent technique. According to Wikipedia, “fuzz testing was developed at the University of Wisconsin-Madison in 1989 by Professor Barton Miller and the students in his graduate Advanced Operating Systems class. Their work can be found at [11].”

An open question is: “what are the differences between classic software testing techniques and fuzzing?” as noted in [12].

It is really hard to answer: the final goals are probably different. Today fuzzing is mainly focused at finding implementation bugs that will result in security flaws, whereas classic software testing techniques are focused on interoperability, good operation of implemented features, memory leaks and so on…

In this paper, we will consider that fuzzing is a software testing technique aiming at finding implementation bugs without imposing any particular testing strategy (random, valid, malformed/semi-malformed…).

3.2 Fuzzing: a good choice?

The main interest of fuzzing is its price-earning trade-off. A basic fuzzer should be easy to implement—e.g. just feeding the fuzzed application with random data—and would quickly find the most obvious implementation bugs. Of course, it is not feasible to test every code path with a fuzzer as testing space is virtually infinite (and testing time is unfortunately finite). Fuzzing is highly recommended as a first shot tool against an application in order to find the most obvious bugs if the fuzzer is protocol-oriented (or application-oriented), but this technique cannot be helpful for identifying complex bugs that could be found by source code auditing or reverse engineering only.

Fuzzing may be considered as a part of the software testing domain (since they have different goals, all testing techniques are indeed complementary).

3.3 Some fuzzing successes

Fuzzing is now a popular technique when finding security vulnerabilities. A lot of vulnerabilities are identified thanks to fuzzing research as it is usually performed by some software vendors (e.g. Microsoft) or other initiatives like “Month of Browser Bugs (MOBB)”, “Month of Kernel Bugs (MOKB)”, “Month of Apple Bugs” and “Month of PHP Bugs” [13–16] that published a new security vulnerability every day during 1 month.

The file system fuzzer *fsfuzzer* [17] released during the MOKB was very effective. His author disclosed numerous

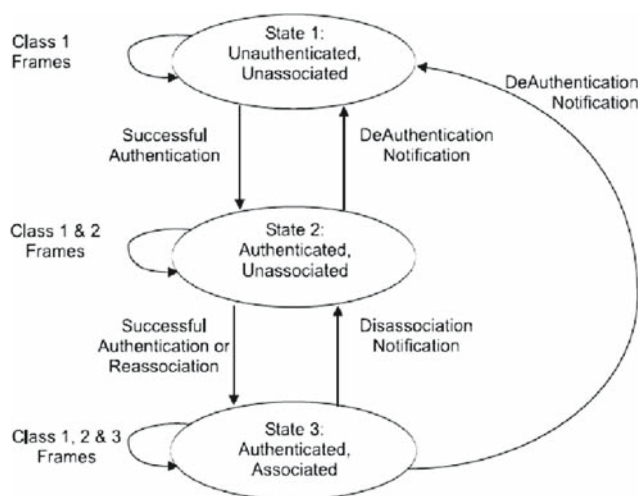


Fig. 1 802.11 Finite state machine

critical security flaws in most file systems[2] thanks to a basic fuzzer that randomly mangles some bytes in the file to be mounted—at the source code level, there is no tests aiming at a specific file system. So even a basic fuzzer may be extremely effective, but, when considering the finding of complex bugs, fuzzing is much less efficient.

4 802.11 Standard overview

4.1 802.11 Finite state machine

The 802.11 finite state machine exposed in Fig. 1 are composed of three kind of states:

- State 1: initial state, not authenticated, not associated;
- State 2: authenticated,[3] not associated;
- State 3: authenticated, associated.

For 802.11 wireless communications, the authentication, association, deassociation and deauthentication procedures enable the wireless client and access point to be synchronized regarding their own finite state machines. De-authentication and deassociation procedures are in charge in keeping the state machines synchronized.

From client-side, we have:

- State 1 is the initial state where the client device probes for access points,
- State 2 is the authenticated state where it is authenticated to the access point,
- State 3 is the associated state where it is authorized to send (and receive) data communication frames to (and from) the wired network through the wireless access point. All state transitions are processed thanks to 802.11 management frames.

The 802.11 standard specifies the state machine as it must be implemented in firmware and/or driver, but of course, numerous other internal states in the driver are managed in order to fully operate. For example, an access point will accept association requests that includes a valid configured network name only. Regarding the implementation, numerous states must be implemented in the driver, and that is implementation dependent…This is a critical issue for the fuzzing process as the purpose is to test as many as possible different code paths in order to maximize the code coverage by choosing an adequate testing strategy. This is an important requirement for the design of an effective fuzzer.

4.2 Scanning for 802.11 networks

The 802.11 standard [1] defines two different scanning mechanisms in order to detect wireless access points:

- Active scanning: a wireless client sends successively probe request frames to the broadcast address on every 802.11 channel and listen to beacon and probe response frames sent back by access points (they must answer to such probe request frames if it contains a null SSID—also called empty SSID, which can be translated by any SSID—or the configured SSID network name), these frames are advertising all required information (network name, channel, rates, cryptographic capabilities) in order to show all these to the user through a user interface;
- Passive scanning: wireless client listens to beacon frames on every 802.11 channels successively (beacon frames are usually sent periodically[4] by access points), these frames are advertising all required information (network name, channel, rates, cryptographic capabilities) in order to show all these to the user through a user interface.

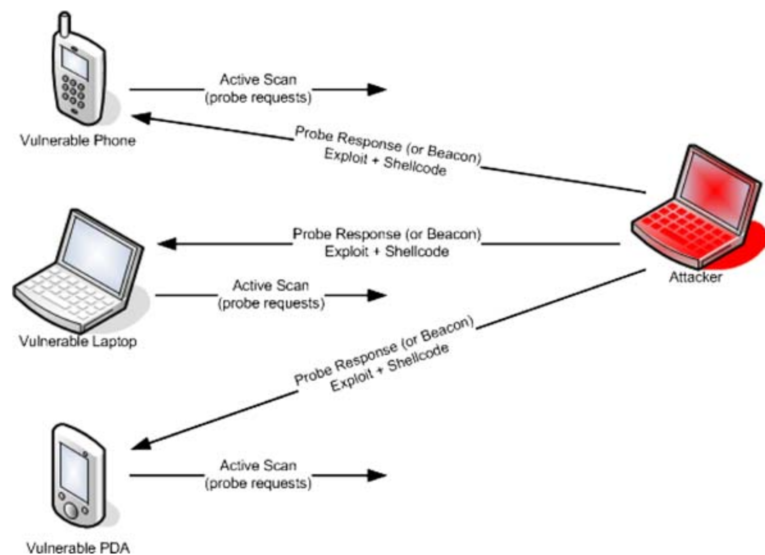
Thanks to these two scanning techniques, the wireless client is able to build a list of nearby access points. Wireless Zero Configuration service on Microsoft Windows and the *iwlist* tool of *wireless-tools* [18] under Linux are using these scanning techniques. These two methods are critical for 802.11

[2] We speak of file systems that are parsed and thus the parser may have implementation bugs as well; and as the parser is located in kernel-land because mounting file systems is devoted to the kernel, the vulnerability are somewhat critical.

[3] In the 802.11 sense, i.e. “open” or “shared key” authentication, which is completely different than security mechanisms implemented in the 802.11i standard.

[4] Usually a period of 100 ms is used.

Fig. 2 802.11 Station attack overview



fuzzing as the first step in 802.11 client-side fuzzing will be devoted to send back appropriate beacon and probe response frames in order to fuzz the initial state (i.e. detecting access points). Of course, this is the most promising state to fuzz as any exploitable bug would be exploited whenever the client scans for wireless networks (i.e. without needing to be associated to an access point—only by switching the wireless interface on). Regarding security mechanisms, we must also notice that some 802.11 firmware/driver implementations may behave differently from the 802.11 standard and this will have impacts on the design of the fuzzer. This particular issue will be discussed in the next part of our paper.

5 Design of a 802.11 fuzzer

5.1 Which frames to fuzz?

Every 802.11 states can be fuzzed. However, fuzzing state 1 is obviously easier than fuzzing state 2 and 3; as a matter of fact, fuzzing other states requires a successful process of authentication and/or association. Regarding the 802.11 MAC layer, the acknowledgment of 802.11 frames is a requirement for the frames to be accepted at MAC layer; thus for a successful authentication, every frame must be ACKed within a short period of time. This is usually hardly feasible efficiently by user-land processes as acknowledgments procedures are performed by the firmware. To fuzz other states, a solution for ACKing 802.11 frame is mandatory while in monitor mode.[5]

[5] This is a specific mode on some chipset, firmware and driver implementations that allows arbitrary frame injection which is a requirement for fuzzing, as we want to inject any kind of wireless frame (even if not compliant to the 802.11 standard)!

In this paper, we will detail our work on state 1 client side fuzzing as shown in Fig. 2. If exploitable security bugs are found, then it implies possible compromising during the scanning phase which is extremely critical. It is not necessary for the client to be associated to a rogue access point to be exploited: as the scanning is usually performed whenever you activate the wireless interface, your kernel (and then your operating system) can be compromised by listening to one malicious beacon or probe response frame.

The risk is extremely critical comparing to other classic wireless attacks that heavily rely on rogue access points [19] or wireless traffic injection in upper layer communications (HTTP...) [20,21]. Thanks to wireless driver vulnerabilities, any exploitable security flaw will give the attacker the capability to execute arbitrary code with kernel privileges.

Today the popularity of 802.11 technology is obvious, 802.11 enabled devices are now ubiquitous: laptops, cellular phones, PDAs, printers, cameras…So many potential vulnerable devices and so hard to mitigate security issues!

5.2 Main constraints for a 802.11 fuzzer

This chapter summarizes some issues encountered during the design of our client-side 802.11 fuzzer.

802.11 state machine fuzzing Fuzzing the association procedure of a client to an access point is much more complicated than only fuzzing the initial state. As a matter of fact, some timing constraints at the 802.11 MAC layer level impose that 802.11 frames must be acknowledged with 802.11 control frames within very short delays—about 300 µs —and thus making it very difficult to emulate it in user-land under monitor mode of a wireless card. So during the association process, the fuzzer would be able to send both authentication

and association responses but also acknowledgment frames for authentication and association requests (emulating a true access point), if not the case, its responses will not be processed by the client device. We encountered this kind of issue when developing a proof-of-concept tool called Raw Glue AP [22] that emulates a set of virtual access points in monitor mode, in order to capture any wireless clients in scanning mode (a kind of captive portal at 802.11 MAC layer).

That's one of the reason why we implemented only the state 1 fuzzing. Moreover, most implementations when they check for correct length or values in frames are calling the same functions for both checking probe responses or association responses (for example): thus, it is more than probable than finding an implementation bug in state 1 will be the same case than finding it in state 2. But, of course, this is only an assumption from a developer point of view, and real life can be quite different.

Firmware fuzzing This article focuses on 802.11 drivers only. But some 802.11 chipsets are managing the MAC layer in their firmware and are not entirely relying on the driver for the MAC layer. For example, Prism2.5 and Prism54 chipsets are parsing probe request and probe response frames in the firmware, so, if an implementation bug is found, it is likely to have some impact on the wireless connexion (e.g. hanging and/or resetting the firmware) [25].

The main difficulty is to find this implementation bug as there is usually no verbose information from the firmware to the operating system (no logging, no debugging…), most of the time the driver may detect that the firmware is non responsive and may reset it. The second issue would be to exploit this vulnerable firmware! So, if such issues are identified, these should only imply denial of service attacks.

Test space versus testing time When considered random, the entire test space is virtually infinite as every byte represent a range of 256 possibilities, so full random fuzzing has nearly no chance to be effective if the protocol is not trivial. Protocol-oriented fuzzing is a better approach in order to elect only interesting fields that will be parsed by the driver (this is the goal of an efficient fuzzer, try to choose tests that will be parsed by multiple parts of the driver). If the fuzzer implements a smart testing strategy which is designed according to the protocol, it will improve the overall code coverage. Thus, it is interesting to choose the best options to fuzz some specific fields, for example, fuzzing strings with format strings, fuzzing bytes and words with boundary values in order to trigger overflows…The design of the fuzzer must take into account the most relevant tests by imagining which programming errors would have been done in a buggy driver.

To the opposite, the 802.11 fuzzer disclosed at the Black Hat US 2006 [23] conference considered a completely different approach as it is a fully random testing strategy.

These two different techniques may be complementary but we preferred to implement a protocol-oriented version for both testing time requirements—1 day of full testing and no more—and efficiency, i.e. finding bugs.

Force 802.11 scan At first sight, this seems to be obvious, but it is necessary to be sure that the wireless driver is parsing the frames sent by the fuzzer. So it is more than interesting to force the scanning process in order to accelerate tests[6] and reduce false negatives.[7] This is implementation-dependent.

Under Microsoft Windows, we force 802.11 scanning thanks to NetStumbler [24], and under Linux, thanks to *iwlist* that calls `SIOCSIWSCAN` and `SIOCGIWSCAN` when executed in privileged mode (`SIOGIWSCAN` only when executed in non privileged mode, in this particular case, scanning results are given back thanks to a background scanning—in the madwifi driver, it is the *bgscan* value in its configuration—that calls `SIOCSIWSCAN`).

Making sure that packets are parsed by the driver During scanning, the 802.11 wireless interface listens to every channel during a given time, which depends on its capabilities and configuration. When tests are sent on a different channel than the wireless card listens to, it could lead to false negatives as these packets will not be interpreted and parsed by the driver. A good workaround is to flood the radio with both beacon and probe response frames during several seconds in order to be sure that the particular packet will be parsed by the driver.

Of course, you must be sure that the wireless card is still in scanning mode during all the fuzzing process (in case of it associates to an open access point) thanks to functions in the fuzzer devoted to check if active scanning is still operating (e.g. by listening to probe request frames sent by the wireless device).

Finding implementation bugs During the fuzzing process, malfunctions may occur at the 802.11 driver level—in fact, it is strongly suggested! Under Microsoft Windows, critical implementation bugs will trigger a Blue Screen of Death (and optionally[8] a memory dump). In this particular case, it is quite easy to detect it as the station is neither responding anymore to any request (at network layer, e.g. wired Ethernet) nor have any wireless activity, this will enable us to stop on the latest test that triggered the bug.

Under Linux, it is quite different, as driver crashes or bugs will be logged thanks to the kernel logging features (BUG, oops…). Thus, it is interesting to parse the kernel logs in order

[6] I.e. we do not want to wait for the driver to operate 802.11 scans regularly, that are usually defined by default values especially in embedded devices like 802.11 enabled cellular phones.

[7] Fuzzing tests that will not be parsed by the driver.

[8] It is configurable.

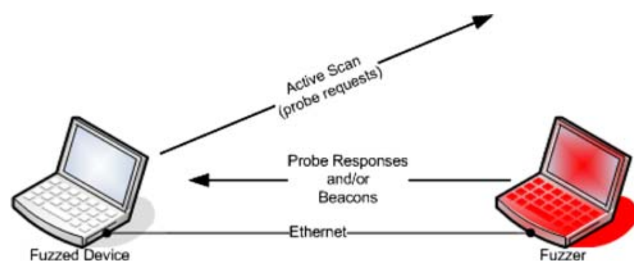


Fig. 3 802.11 Fuzzing architecture

to match some interesting keywords and stop the fuzzer on the latest test that triggered the implementation bug.

Other techniques are possible, especially when fuzzing 802.11 devices that do not have log nor wired interface such as 802.11 enabled phones. For example, listening wether the active scanning process is still operating by listening to probe requests sent by the wireless device: if no more probe requests occur, it is likely to have crashed. This particular technique works for any kind of device.

Of course, in this part, we are finding only critical bugs that have an impact on normal operation of the wireless device. A lot of implementation bugs are still very hard to trigger thanks to fuzzing and crash detection techniques. This is unfortunately quite hard to overcome, especially when testing embedded devices. For example, some information leak bugs were discovered by our fuzzer, when it was possible to make the driver read some adjacent memory zones (of the 802.11 packet), but as these memory areas are readable this is not a major malfunction (e.g. a buffer overflow) and thus cannot be detected by the crash detection techniques exposed in this chapter.

In Fig. 3, the Ethernet interface enables us to detect driver malfunctions in order to stop the fuzzing process on the latest test that triggered the implementation bug. This is complementary with the wireless interface that listens to probe requests in order to check whether the scanning process is still operating or not. It is also possible to attach a kernel debugger on the fuzzed device communicating through the wired interface, in order to catch exceptions and to trace the driver. But of course, this is only feasible on general purpose operating systems, i.e. not on embedded devices like 802.11 enabled phones.

5.3 Implementation

Chipset for wireless injection The Atheros chipset and its Open Source driver madwifi [26] under Linux are well known within the hacking community as they are very flexible—the firmware is minimalist and the MAC layer is operated in the driver. As a matter of fact, fuzzing requires to tweak all protocol fields that could be operated (and thus overwritten) by the firmware. It is critical to use a chipset that let us the capability to perform arbitrary wireless frame injection over 802.11 bands (these frames could be compliant or not to the 802.11 standard). One could imagine that 802.11 protocol conformance may be enforced at firmware level, nowadays it is not the typical case since most chipset manufacturers prefer to implement the MAC layer in software rather than in the chipset, probably for cost reasons and flexibility.

Some fields are usually overwritten by most—but not the Atheros one—firmwares as:

- The sequence number value in management and data frames,
- The BSS Timestamp value in beacon and probe response frames,
- The duration ID value,
- The capability to inject fragmented frames,
- The capability to inject control frames.

The Atheros chipset supports all above-mentioned features except the sequence number forging that can be activated thanks to a small patch [27] (but will set the retry flag to 1 for every injected frame which could be bad from an attacker point of view; indeed retries will be detected by a wireless intrusion detection system as an abnormal traffic). Taking a look at LORCON [28] source code (which is a library for wireless device abstraction and frame injection) gives better information on chipset capabilities regarding monitor mode, frame injection…

Programming language Even if it is possible to write a fuzzer in C/C++ languages [23], it is more convenient to use a high level language for 802.11 packet forging. We have chosen the Python language for its capabilities, ease of use and good performance.

As we will flood the radio with both beacon and probe response frames, there will be no issue regarding frame injection performance where it would be necessary to answer to probe request frames within a small window (several hundreds of microseconds). As a matter of fact, probe request frames sent by the client do not contain any cookie necessary for the probe response (and thus emulating a kind of session for the scanning process). So flooding the radio with both beacon and probe response frames is by far the best option (effective and easy to implement). Moreover, as stated in the Johnny Cache, HD Moore and skape paper [29], some drivers are only parsing beacon frames if probe response frames are also sent in the air. Flooding the radio, seems again the best option!

Relying on existing tools or development from scratch? In our first version of the fuzzer, we developed a set of functions to forge and inject 802.11 packets. Now, we have

implemented our fuzzer thanks to Scapy [30] that implements most parts of the 802.11 protocol and load of useful functions for sending and receiving frames.

5.4 Choosing the 802.11 fields to fuzz

This section details the most important 802.11 frames. A good understanding of the protocol is a requirement to identify the best fields to be fuzzed. During the design of the fuzzer, it is mandatory to be aware of possible implementation bugs and to find means to trigger them by tuning the most appropriate 802.11 fields.

In Table 1, the frame control defines the 802.11 frame, other fields in Table 1 do not seem to be promising regarding fuzzing.

In Table 2, the frame control defines the 802.11 frame header. According to type and subtype values, the frame could be a management frame (beacon, probe request…) or any other frame. The type and subtype values are defined in the 802.11 standard and its extensions (QoS, action frames…). It may be interesting to fuzz these fields since the driver needs it to identify the kind of frame in order to parse the next layer (which is defined thanks to the standard). Fuzzing at this level checks the preliminary parts of the driver whenever it must check for the frame type to be parsed or dropped.

In Table 3, management frames have a predefined frame control (according to 802.11 standard). For example, it defines that no more than three MAC addresses are set in this mode. The driver parses the subtype value in order to parse differently probe request and beacon frames for example. Fuzzing at this level extends the code coverage comparing to the latter test.

In Table 4, beacon and probe response frames are particularly interesting as they are parsed by wireless drivers during 802.11 scanning. The timestamp, beacon interval and capability fields are mandatory but do not seem to be promising regarding implementation bugs (e.g. overflows or format strings).

In Table 5, information elements are optional fields after mandatory fields in management frames. The information elements are specified by type, length, and value attributes.

Table 1 802.11 Frame

Field	Size	Information
Frame control	16 bits	802.11 Frame identification and attributes
Duration/ID	16 bits	NAV update and/ or association ID
Address 1	48 bits	First MAC address
Address 2	48 bits	Second MAC address
Address 3	48 bits	Third MAC address
Sequence control	16 bits	Sequence and fragmentation numbers
Address 4	48 bits	Fourth MAC address (WDS mode[a])
Frame body	0–2312 octets	Frame body
FCS	32 bits	Integrity check (CRC32)

[a] Wireless distribution system: repeater mode over wireless extend the wireless coverage thanks to WDS enabled access points

Table 2 802.11 Frame control header

Champ	Size (bits)	Information
Protocol	2	For the 802.11 standard, equals 0
Type	2	802.11 Frame type
Subtype	4	802.11 Frame subtype
To DS	1	To distribution system
From DS	1	From distribution system
More frag	1	802.11 Fragmentation
Retry	1	Retransmitted 802.11 frame
Pwr Mgt	1	Save energy mode announced by stations
More data	1	802.11 Frames ready to be sent from access point
WEP	1	Set to 1 if frame body encrypted
Order	1	Set to 1 if strictly ordered activated

Table 3 802.11 Management frame header

Field	Size	Information
Frame control	16 bits	802.11 Frame identification and attributes
Duration/ID	16 bits	NAV update and/or association ID
Address 1	48 bits	First MAC address
Address 2	48 bits	Second MAC address
Address 3	48 bits	Third MAC address
Sequence control	16 bits	Sequence and fragmentation numbers
Frame body	0–2312 octets	Frame body
FCS	32 bits	Integrity check (CRC32)

Table 4 802.11 Beacon or probe response header

Information	Size (bits)	Type
Timestamp	64	Synchronizing clock
Beacon interval	16	Time between two beacons
Capability information	16	Access point capabilities

Table 5 802.11 Information element

Field	Size	Type
Type	1 byte	Information element type (ID)
Length	1 byte	Information element length
Value	Length byte(s)	Information element value

This is extremely promising regarding implementation bugs! The Length value may be useful in order to trigger buffer overflows if not carefully checked by the implementation during the parsing of the 802.11 frame. Most of these information elements have minimal, fixed or maximal values that can be different from byte boundaries (0 and 255). Thus, implementation bugs in this part of the driver code are more than probable.

In Table 6, the most obvious implementation bug we want to discover is the allocation of a 32 bytes static buffer for the SSID information element, then reading the advertised length in the information element, and lastly copying the information element payload in the static buffer without checking the copied length, which may lead to a buffer overflow. Many other bugs are possible (NULL pointer dereference, integer overflows…) by invalid parsing the information elements: they must be extensively fuzzed!

5.5 Implemented features

During the design of the fuzzer, we implemented several testing strategies. We focused on the information elements fuzzing as they were really promising in terms of possible implementation bugs (remember that Type, Length, Value fields are error-prone). These tests are then skipped till no further test is available or if a bug is found.

Implemented testing strategies are:

- Optimized information elements fuzzing,
- Brute force information element fuzzing,
- Random information elements fuzzing,
- Partial protocol aware information elements fuzzing,
  - For proprietary information elements (Atheros, Cisco…),
- Full protocol aware Information Elements fuzzing,
  - Wi-Fi Protected Access (`WPA`), Robust Security Network (`RSN`), Wireless Multimedia Extensions (`WMM`)…
- discovered vulnerabilities tests (according to the current state-of-the-art).

These tests may be manually passed by the user of the fuzzer in order to check the most appropriate tests to execute according to the available time for fuzzing.

Table 6 Some information elements

Information element ID	Information element type
0	Service set identifier (SSID)
1	Rates
2	FH parameters
3	DS parameters
4	CF parameters
5	Traffic information map
7	Country
48	RSN
50	Extended rates
221	Vendor specific or WPA

6 Fuzzing in two easy steps

Our fuzzer has advanced features especially regarding testing strategy, but according to the state of the art of 802.11 driver vulnerabilities, most implementation bugs would have been discovered thanks to basic fuzzers. Thus, this chapter explains how to perform 802.11 fuzzing at an incredible low-cost.

6.1 Fuzzing with scapy

The `fuzz()` function generates random values for non filled fields during the frame specification by the user. It is quite trivial to implement a basic 802.11 fuzzer thanks to *Scapy* [30].

Random fuzz

```
frame = Dot11(addr1=DST,addr2=BSSID,addr3=BSSID,addr4=None)
sendp(fuzz(frame), loop=1)
```

Sent frames are both valid and invalid 802.11 frames according to the standard, and most of them will be considered as malformed frames. This test evaluates the capability of the driver to drop invalid and malformed frames.

Information elements random fuzzing in beacon frames

```
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()
sendp(fuzz(frame), loop=1)
```

Sent frames are only beacon frames with random information elements that are not necessarily specified in the 802.11 standard. This test evaluates the capability of the driver to parse both valid and invalid information elements in a valid frame (beacon).

SSID information element random fuzzing in beacon frames

```
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)
sendp(fuzz(frame), loop=1)
```

Sent frames are only beacon frames with SSID information element with different lengths that are not correct regarding the 802.11 standard. This test evaluates the capability of the driver to parse both valid and invalid SSID information elements in a valid frame (beacon).

6.2 Fuzzing with metasploit

Thanks to the LORCON [28] library integration in Metasploit [31] framework, it becomes easy to create and inject arbitrary 802.11 frames within Metasploit modules. There are some interesting examples in the SVN that aim at creating fake access points or executing fuzzing tests.

Randomly fuzz probe responses information elements

```
./msfcli    auxiliary/dos/wireless/fuzzproberesp
            DRIVER=madwifing ADDR_DST=11:22:33:44:55:66
            PING_HOST=192.168.1.10
            E
```

These probe response frames include information elements that are not necessarily correct in the 802.11 standard point of view. This particular fuzzing technique aims at testing the driver capabilities in parsing both correct and incorrect probe response frames. Moreover the module tries to regularly ping the wireless host in order to check whether it is still alive and then stop the last test in case of the fuzzed device has crashed.

7 Discovered vulnerabilities thanks to our fuzzer

Of course, all tests were performed in order to discover unknown implementation bugs. Thus, we used only latest versions of drivers and firmwares. Sometimes, we found some vulnerabilities that were silently patched by the vendor; thus these vulnerabilities cannot be considered as new or unknown as they were known by the vendor…

Using our fuzzer on several 802.11 driver implementations enabled us to find several critical bugs that would be remotely exploitable and to a minimum would result in a remote denial of service. We focused on the most interesting one only, that we succeeded in exploiting: the first remotely exploitable 802.11 driver under[9] Linux! Moreover, this vulnerability was on the driver of a very popular chipset in the hacking community, i.e. the Atheros chipset.

[9] Note the word "under" is used and not "in", as madwifi is not a part of the mainline kernel.

7.1 Netgear MA521 wireless driver long rates overflow (CVE-2006-6059)

This vulnerability was disclosed on November 18, 2006 thanks to Month of Kernel Bugs [32].

An overflow on the "RATES" information element triggers the bug (Blue Screen of Death on Microsoft Windows). According to the kernel dump, it seems to be a heap overflow.

We did not investigate further and we published the Metasploit module for the denial of service only. To date, no security fix is available yet.

7.2 Netgear WG311v1 wireless driver long SSID overflow (CVE-2006-6125)

This vulnerability was disclosed on November 22, 2006 thanks to Month of Kernel Bug [33].

An overflow on the "SSID" information element triggers the bug (Blue Screen of Death on Microsoft Windows). According to the kernel dump, it seems to be a stack overflow.

We did not investigate further and we published the Metasploit module for the denial of service only. To date, no security fix is available yet.

7.3 D-link DWL-G650+ wireless driver long TIM overflow (CVE-2007-0933)

This vulnerability was disclosed on March 28, 2007 thanks to Black Hat Europe conference [34,35].

An overflow on the "TIM" information element triggers the bug. According to the kernel dump, it seems to be an off-by-one stack overflow.

We did not investigate further and we published the Metasploit module for the denial of service only. To date, no security fix is available yet.

7.4 Madwifi driver "giwscan_cb()" and "encode_ie()" remote buffer overflow vulnerability (CVE-2006-6332)

This vulnerability was disclosed on December 7, 2006 thanks to Daily Dave [36]. We reported the vulnerability to the Madwifi team on December 5, 2006 and they issued a patched release with an advisory on their website on December 6, 2006. So, we waited for the public announcement before publishing a proof-of-concept exploit that was only a "local" exploit (which was not critical because it required a local user account to be exploited).

We thank the Madwifi team for their reactiveness as they released a patched version with all relevant information to their users the day after we contacted them. Overall security level of a software is generally a matter of process (in particular when managing security incidents), and, here this is a pretty good example of software vendors would do. Because,

implementation bugs are hard to avoid in complex software, so, the main issue is to manage the correction of security bugs with efficiency (publishing an advisory, releasing patches and new stable version…).

An overflow on the `WPA`, `RSN`, `ATH` or `WMM` information elements triggers the bug (kernel oops). According to the kernel oops, it is a stack overflow.

8 Vulnerability exploitation: a practical case

In this chapter, we now detail where the vulnerability is located and how to trigger it with a specifically crafted 802.11 frame. We must not forget that only one 802.11 frame may perform remote arbitrary code execution in kernel mode over a 802.11 radio link (from an unauthenticated attacker). It is indeed a quite critical issue.

8.1 Introduction

Our fuzzer implements some features designed to fuzz particular information elements like `WPA` and `RSN`…Thanks to these features, the fuzzer has triggered the driver by fuzzing `WPA` information elements.

The driver parses all information elements included in the beacon or probe response frames in order to detect `WPA` information elements for further parsing—the cryptographic capabilities such as authentication method or multicast and unicast encryption protocols. For the `WPA` information element to be parsed, it must contain a specific header which is composed of `OUI` (Organizationally Unique Identifier), `TYPE` and `VERSION` fields that accordingly fit the driver requirements (according to its configuration).

So a basic fuzzer is not be able to discover this kind of implementation bug because the `WPA` header must be (to a minimum) correctly filled by the fuzzer. A minimal understanding of the protocol must be implemented in the fuzzer. That is the reason why this vulnerability was not detected sooner than we did! Moreover the vulnerability is located in an Open Source driver which could be easily spotted by static analysis or code reviewing (even if quite time consuming).

This vulnerability is also present in `RSN`, `WMM` and `ATH` information elements parsing.

```
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)
```

Analyzing this Linux kernel "oops" enabled us to detect that several registers where overflowed by the attacker, and more precisely a well-known register: `EIP`! If the attacker masters the content of `EIP` register, it makes possible remote arbitrary code execution. Thus, one the main goals would be to point `EIP` directly or indirectly to interesting instructions that are mastered by the attacker (i.e. in the 802.11 frame that triggered the vulnerability).

8.2 Multiple vulnerabilities

In `giwscan_cb()` function

```
static void
giwscan_cb(void *arg, const struct ieee80211_scan_entry *se)
{
    struct iwscanreq *req = arg;
    struct ieee80211vap *vap = req->vap;
    char *current_ev = req->current_ev;
    char *end_buf = req->end_buf;
#if WIRELESS_EXT > 14
    char buf[64 * 2 + 30];
#endif
```

All recent Linux kernels are compiled with wireless extensions greater than 14, thus a static buffer `buf` of 158 bytes is allocated.

```
<snip>

#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;
```

If the `IWEVGENIE` parameter is defined, as it is the case in recent Linux kernels, a first vulnerability is present when `memcpy()` is called with a non checked length. The static buffer `buf` is 158 bytes length, and `se->se_wpa_ie[1]` may have the value of 255 (as copied from the `WPA` information element in the 802.11 frame). This value is never checked in any other part of the driver, making it possible for the attacker to trigger the stack overflow putting a maximum of 257 bytes in a 158 bytes-long buffer!

```
#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
```

If the `IWEVGENIE` parameter is not defined, the `encode_ie()` function is called with arguments that could trigger the bug. If a second vulnerability exists, it is located in `encode_ie()` fuzzing.

```
<snip>

    if (se->se_wme_ie != NULL) {
        static const char wme_leader[] = "wme_ie=";

        memset(&iwe, 0, sizeof(iwe));
        iwe.cmd = IWEVCUSTOM;
        iwe.u.data.length = encode_ie(buf, sizeof(buf),
            se->se_wme_ie, se->se_wme_ie[1] + 2,
            wme_leader, sizeof(wme_leader) - 1);

<snip>

    if (se->se_ath_ie != NULL) {
        static const char ath_leader[] = "ath_ie=";

        memset(&iwe, 0, sizeof(iwe));
        iwe.cmd = IWEVCUSTOM;
        iwe.u.data.length = encode_ie(buf, sizeof(buf),
            se->se_ath_ie, se->se_ath_ie[1] + 2,
            ath_leader, sizeof(ath_leader) - 1);

<snip>
```

With `WMM` or `ATH` information elements, it is the same operation than when the `IWEVGENIE` parameter is not defined, i.e. the `encode_ie()` function is called with arguments with non checked sizes. If another vulnerability is present, it may be located in the next function called.

In `encode_ie()` function

```
static u_int
encode_ie(void *buf, size_t bufsize, const u_int8_t *ie,
    size_t ielen, const char *leader, size_t leader_len)
{
    u_int8_t *p;
    int i;

    if (bufsize < leader_len)
        return 0;
    p = buf;
    memcpy(p, leader, leader_len);
    bufsize -= leader_len;
    p += leader_len;
    for (i = 0; i < ielen && bufsize > 2; i++)
        p += sprintf(p, "%02x", ie[i]);
    return (i == ielen ? p - (u_int8_t *)buf : 0);
}
```

The `ielen` variable is controlled by the attacker and `p` is a pointer to the static buffer `buf`. The vulnerability is then triggered when `sprintf()` is executed since the information element is converted in ASCII in `buf`. The main issue about possible exploitation is that the information element is converted in ASCII and so the exploit is also converted!

8.3 Remote exploitation

Given the control of the instruction pointer (`EIP`), we now aim at detecting an address whose code is under our control. The obvious way to inject code into the process' address space[10] is to use the very 802.11 frame that triggers the vulnerability.

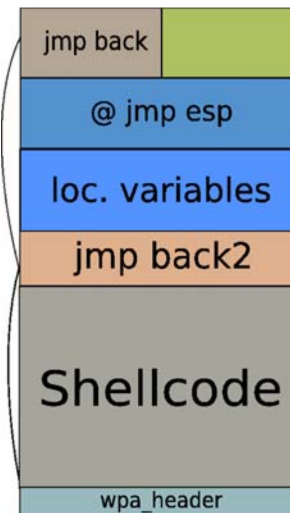


Fig. 4 Kernel stack

We should have enough space to embed interesting shellcodes, because `buf` is 158 bytes long. Only the first six bytes are not usable because they identify the information element as `WPA` (`OUI + TYPE + VERSION`).

Our problem is now to locate our 802.11 frame, which embeds our shellcode, in memory. We know that it will be on the kernel stack, so our idea is to find a `jmp esp` with a stable, fixed, address and make `EIP` point to it. That way, we will execute an instruction in the 802.11 frame that we control, this instruction will be another jump that will lead to the beginning of the information element, where our real shellcode lies. The reason behind this second jump is that we do not want to "crash" the kernel stack. The `jmp esp` will execute the instruction on the kernel stack that lies just after the saved instruction pointer (that we overwrite to control the execution flow).

The problem is that after the saved instruction pointer there are valuable data such as function arguments, local variables or saved registers that we do not want to overwrite. That is the reason why we encode the smallest possible jump (2 bytes) and jump back to the beginning of the information element. Unfortunately, it is too far away for a 2-bytes jump (more than 128 bytes), and we will have to proxy it, one more time, through a third jump as shown in Fig. 4.

We now are able to remotely execute arbitrary code in kernel mode on any Linux with a vulnerable version of Madwifi. Most of our tests took place on a stock 6.10 version of the Ubuntu Linux distribution (with a version of the restricted modules packages that includes a vulnerable madwifi version).

[10] We are in process context, since we are in a system call.

The last part is to write a kernel-mode shellcode. As we are in process context, it is really very easy to get back to user-land and execute an ordinary user-land shellcode. For instance, we can copy a user-land shellcode to the user-land stack, temper with the saved user-land instruction pointer (`EIP`) which lies on top of the process' kernel stack and make it point to the user-land stack. We now only have to return to user-mode.

The easy way would be to do an `IRET`, but this would leave the 802.11 stack in a flimsy state. If we want to do this properly, we have to emulate the epilogue of the function that called the vulnerable function (we cannot just return to it because we would overwrite the return address) and return to the caller of the caller. That way, we will ultimately return to user-land and execute our user-land shellcode without breaking the 802.11 stack. There are however some efforts involved in the caller's epilogue emulation, because there are spinlocks to unlock.

We have written an implementation of this exploit for the Metasploit[31] framework which is now available (for the interested reader) in the SVN repository.

8.4 Triggering the vulnerability

The vulnerability is located in the `giwscan_cb()` function. This function is called thanks to a `SIOCGIWSCAN`. This system call enables the analysis of a wireless scan and to show them to the user (e.g. thanks to a command-line interface). For example, the *iwlist* tool calls this system call thanks to `iwlist ath0 scanning` with non privileged rights. But, for the vulnerability to be exploited, it is mandatory that malicious frames are given through the structure `const struct ieee_scan_entry *se` which is possible only if a 802.11 scan has updated the wireless scanning data:

- Manually thanks to a `SIOCSIWSCAN` (e.g. with `iwlist ath0 scanning` with privileged rights),
- Automatically by the 802.11 driver when "background scanning" occurs (tunable with internal parameters like `bgscan`, `bgscanidle` and `bgscanintvl`),
- Automatically thanks to third-party tools like *wpa_supplicant* that performs scanning periodically.

9 Automated exploitation

Thanks to 802.11 fingerprinting techniques, it would possible to elect the best appropriate exploit regarding the wireless device. So, automated exploitation is more than feasible especially in public area networks like conferences or hot spots.

Metasploit, the infamous exploitation framework, now includes LORCON and its ruby bindings making it possible to exploit any vulnerable wireless driver (note that several Windows-based 802.11 driver exploits are available). At the present time, our madwifi remote exploit is now included in the Metasploit SVN repository.

10 Automatic spreading of malware thanks to wireless driver vulnerabilities

Since it is possible to remotely execute arbitrary code with kernel privileges, it is potentially feasible to propagate a piece of code such as a Trojan dropper that will compromise the victim (e.g. download and execute function). But, the main issue is to keep the network stack (802.11 and/or Ethernet) intact during the exploitation, this is not an easy task and depends on the vulnerability, driver and kernel.

For example, in the madwifi exploitation, we have succeeded in releasing properly the spinlocks giving the possibility to execute commands that will rely on network (like a reverse connection) without smashing the 802.11 stack. Thus it is possible to perform a reverse connect to an Internet host if the victim is associated to a IP network over a 802.11 link. But, it is really hard to have an autonomous malware spreading through wireless exploits as the main requirement is to inject 802.11 arbitrary frames (which is not available in out of the box drivers running on Microsoft Windows and which requires the driver to use monitor mode).

To sum up, wireless driver vulnerabilities may be useful to compromise hosts and install some kind of malicious code (and then for a targeted attack), but are really hard to use for an autonomous propagation relying only on 802.11 wireless networks (due to 802.11 frame injection limitations in usual configurations).

11 Conclusions

In this paper, we have undeceived 802.11 driver vulnerabilities. Most publicly disclosed vulnerabilities were easy to spot and a basic fuzzer would have discovered most of the disclosed 802.11 driver vulnerabilities. Our fuzzer enabled us to discover several critical implementation bugs, one of them has been successfully exploited: the first 802.11 remote kernel stack overflow under Linux!

Moreover, one could remind that our fuzzing, despite a "smart" testing strategy, is only focused on 802.11 state 1. Thus a lot of other vulnerabilities could be present in both firmware and drivers at other states. Today, there is no demonstration of such issues, but the design of a stateful 802.11 fuzzer for both client and access points would be an answer.

Acknowledgments We thank Jérôme Razniewski for his research on 802.11 driver vulnerabilities, Yoann Guillot for the incredible `metasm` [39] which was quite useful during the exploit design, Raphaël Rigo for his analysis of Microsoft Windows based vulnerabilities, Franck Veysset and Matthieu Maupetit for their second reading.

References

1. IEEE: local and metropolitan area networks, specific requirements, part 11: Wireless LAN Medium Access Control (MAC) and physical layer (PHY) specifications (1997–1999)
2. Cache, J., Maynor, D.: Device drivers, http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Cache.pdf (2006)
3. Black Hat Conference: http://www.blackhat.com
4. Cache, J., Maynor, D.: Hijacking a MacBook in 60 seconds. http://www.youtube.com/watch?v=chtQ1bcHLZQ (2006)
5. James, W.: Thompson's blog, http://he-colo.netgate.com/archives/00000465.htm (2006)
6. Cédric Blancher's blog: http://sid.rstack.org/blog/index.php/2006/10/02/133-se-paierait-on-notre-pomme (2006)
7. Maynor, D.: Its V-A day.... http://erratasec.blogspot.com/2007/02/its-v-day.html (2007)
8. Lemos, R.: Maynor reveals missing apple flaws http://www.securityfocus.com/news/11445 (2007)
9. Johnny Cache's mail: http://lists.immunitysec.com/pipermail/dailydave/2006-September/003459.html (2006)
10. Wikipedia, Rings: http://en.wikipedia.org/wiki/Ring_%28computer_security%29
11. Miller B.: Fuzz testing of application reliability, http://www.cs.wisc.edu/bart/fuzz/ (1990–2006)
12. Fuzzing Mailing List: The definition of what a fuzzer really is... is fuzzy, http://www.whitestar.linuxbox.org/pipermail/fuzzing/2006-May/000033.html (2006)
13. Month of Browser Bugs: http://browserfun.blogspot.com/ (2006)
14. Month of Kernel Bugs: http://kernelfun.blogspot.com/ (2006)
15. Month of Apple Bugs: http://applefun.blogspot.com/ (2007)
16. Month of PHP Bugs: http://www.php-security.org/ (2007)
17. L.M.H.: fsfuzzer, http://projects.info-pull.com/mokb/fsfuzzer-0.6-lmh.tgz (2006)
18. Jean Tourillhes: Wireless Tools for Linux, http://www.hpl.hp.com/personal/Jean_Tourrilhes/Linux/Tools.html, 1996–2007
19. Dino, A., Zovi D., Macaulay, S.: Attacking automatic wireless network selection, http://www.theta44.org/karma/aawns.pdf (2005)
20. Toast: airpwn, http://sourceforge.net/projects/airpwn (2004–2006)
21. Blancher, C.: wifitap, http://sid.rstack.org/index.php/Wifitap (2005–2006)
22. Butti, L.: Raw Glue AP, http://rfakeap.tuxfamily.org/ (2005–2006)
23. Cache, J.: fuzz-e, http://www.802.11mercenary.net/code/airbase-latest-svn.tar.gz (2006)
24. NetStumbler.com, *NetStumbler*, http://www.netstumbler.com/ (2001–2007)
25. Wright, J., Too, S.O., Kershaw, M.: 802.11b firmware-level attacks, http://802.11ninja.net/papers/firmware_attack.pdf (2006)
26. Multiband Atheros Driver for Wireless Fidelity: madwifi, http://www.madwifi.org/ (2004–2006)
27. Aircrack-ng: madwifi-ng injection patch, http://patches.aircrack-ng.org/madwifi-ng-r1816.patch (2006)
28. Wright, J., Kershaw, M.: Loss of radio connectivity, http://www.802.11mercenary.net/lorcon/ (2006–2007)
29. Cache, J., Moore, H.D.: skape, exploiting 802.11 wireless driver vulnerabilities on windows, http://www.uninformed.org/?v=6&a=2&t=sumry (2006)
30. Biondi, P.: Scapy, http://www.secdev.org/scapy (2003–2007)
31. Metasploit, L.L.C.: Metasploit, http://www.metasploit.com/, 2003–2007
32. Butti, L.: NetGear MA521 wireless driver long rates overflow (CVE-2006-6059), http://kernelfun.blogspot.com/2006/11/mokb-18-11-2006-netgear-ma521-wireless.html (2006)
33. Butti, L.: NetGear WG311v1 wireless driver long SSID overflow (CVE-2006-6125), http://kernelfun.blogspot.com/2006/11/mokb-22-11-2006-netgear-wg311v1.html (2006)
34. Butti, L.: D-link DWL-G650+ wireless driver long TIM overflow (CVE-2007-0933), http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-0933 (2007)
35. Butti, L.: Wi-Fi advanced fuzzing, http://www.blackhat.com/html/bh-europe-07/bh-eu-07-speakers.html#Butti (2007)
36. Butti, L., Razniewski, J., Tinnès, J.: Madwifi SIOCSIWSCAN vulnerability (CVE-2006-6332), http://archives.neohapsis.com/archives/dailydave/2006-q4/0291.html (2006)
37. Madwifi : Release 0.9.2.1 fixes critical security issue, http://kernelfun.blogspot.com/2006/11/mokb-22-11-2006-netgear-wg311v1.html (2006)
38. Tinnès, J., Butti, L.: madexploit.c, http://archives.neohapsis.com/archives/dailydave/2006-q4/att-0298/madexploit.c (2006)
39. Guillot, Y.: metasm, http://www.sstic.org/SSTIC07/programme.do#GUILLOT (2006–2007)