Bringing Software Defined Radio to the Penetration Testing Community

Jean-Michel Picod, Arnaud Lebrun, Jonathan-Christofer Demay

Abstract—The large adoption of wireless devices goes further than WiFi networks: smartmeters, wearable devices, etc. The engineers behind these new types of devices may not have a deep security background and it can lead to security and privacy issues when a particular technology is stressed. However, to assess the security of these devices, the only current solution would be a dedicated hardware component with an appropriate radio interface for each available technology. Such components are not easy to engineer and this is why we developed Scapy-radio, a generic wireless monitor/injector tool based on Software Defined Radio using GNU Radio and the well-known Scapy framework. In this paper, we present this tool we developed for a wide range of wireless security assessments. The main goal of our tool is to provide effective penetration testing capabilities to security auditors with little to no knowledge of radio communication systems.

Keywords—Penetration Testing, Software Defined Radio, Smart Grid, Wireless.

1 Introduction

High-technology industries are increasingly depending on the Internet of Things: advanced metering infrastructures, home and building automation, personal health and fitness monitoring, etc. These new types of devices can be portable, part of a mesh network or just located in a spot where wiring is not an option. For these very reasons, they rely on radio communications to send and receive data.

Historically, except for the overwhelmingly adopted WiFi technology, threats to radio communication systems are confined to individuals with very specific knowledge or in possession of even more specific hardware. However, that is also the reason why reliably assessing the security of a particular wireless technology is no easy task for an independent third party, especially when time or resources are limited.

Nonetheless, with threats to information security becoming more and more sophisticated, these new types of devices need to become part of any security assessment policy. The lack of specific tools, covering both hardware and software aspects, is a problem that is already being tackled: affordable dedicated testing components have begun to emerge. For example, we have *RfCat* [3] for sub-GHz ISM radio bands, *Ubertooth* [4] to work on Bluetooth equipments and *Api-Mote* [5] to stress ZigBee ones.

These are not enough to level the playing field and we argue that we need to go one step further. With high-technology industries regularly coming up with new wireless protocols, we cannot afford to wait for radio communication specialists to design tools such as those previously mentioned. It should be noted that this fact has already been highlighted by the NIST in their guidelines for smart grid cybersecurity [1].

In this paper, we introduce Scapy-radio, a tool that provides effective penetration testing capabilities to security auditors with little to no knowledge of radio communication systems. To be able to carry out a wide range of wireless security assessments, it is designed as a generic wireless monitor/injector tool based on Software Defined Radio using GNU Radio and the well-known Scapy framework.

2 Review of components

Not being confined to a set of wireless protocols is a challenging goal: we need to be able to deal with multiple bands, multiple modulations, multiple bitrates and many different types of network packets. This has lead us to choose a modular solution rather than a monolithic one.

Before digging in the next section into the details of Scapy-radio, we first here briefly present the three main components on which our tool is relying to achieve that goal:

- Software Defined Radio;
- GNU Radio;
- Scapy.

2.1 Software Defined Radio

A radio communication system where the signal-capturing components are software-configurable and the signal-processing components are software-implemented is called a Software Defined Radio (SDR). This is exactly what we need in order to be able to capture and process a broad range of radio signals.

The most notorious opensource-friendly and affordable computer-hosted SDR boards are *HackRF* [6], *bladeRF* [7] and *USRP2* [8]. Because they are full-duplex, dual-channel and they offer large radio spectrum capabilities as well as a great amount of bandwidth, we chose to work with two USRP B210 boards.

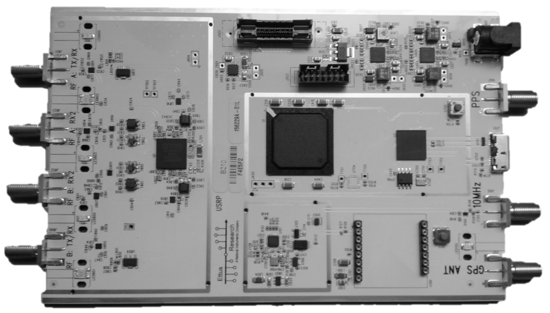
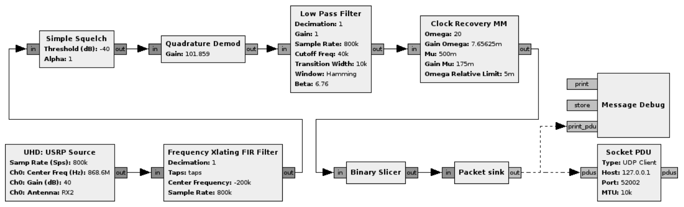


Fig. 1: Ettus USRP B210 board

2.2 GNU Radio

GNU Radio [9] is an opensource software development kit that provides a great number of signal processing blocks to implement SDRs. It is already widely used with the previously mentioned SDR boards but it can also act as a simulation-like environment.

While performance-critical signal-processing blocks are written using C++, GNU Radio is designed to write radio applications using Python. More specifically, radio applications can be prototyped with a graphical UI, the GNU Radio Companion (GRC). We are going to rely on GRC flow graphs to capture signals and turn them into network packets.



Simple Squelch
Threshold (dB): -40
Alpha: 1

Quadrature Demod
Gain: 101.859

Low Pass Filter
Decimation: 1
Gain: 1
Sample Rate: 800k
Cutoff Freq: 40k
Transition Width: 10k
Window: Hamming
Beta: 6.76

Clock Recovery MM
Omega: 20
Gain Omega: 7.65625m
Mu: 500m
Gain Mu: 175m
Omega Relative Limit: 5m

UHD: USRP Source
Samp Rate (Sps): 800k
Ch0: Center Freq (Hz): 868.6M
Ch0: Gain (dB): 40
Ch0: Antenna: RX2

Frequency Xlating FIR Filter
Decimation: 1
Taps: taps
Center Frequency: -200k
Sample Rate: 800k

Socket PDU
Type: UDP Client
Host: 127.0.0.1
Port: 52002
MTU: 10k

Fig. 2: Example of a GRC flow graph

2.3 Scapy

Scapy [10] is an interactive packet manipulation framework written using Python. It can capture, decode, forge and inject packets while matching requests and replies for a broad range of network protocols. It can also handle various network tasks such as probing, scanning, tracerouting, fuzzing, etc.

Because it gives security auditors the capabilities to quickly prototype new networking tools without the need to go into the details of creating raw packets from square one, Scapy is already widely used by the penetration testing community. This is exactly what we wanted to achieve for wireless protocols and that is why we chose Scapy for protocol dissection and user interaction.

3 Software Architecture

Figure 3 shows how all the previously mentioned components interact with each other. In this section, we talk about the software development that was necessary to make that happen.

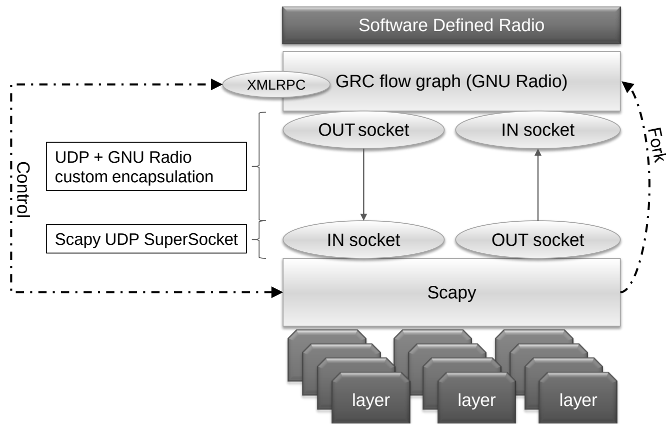


Fig. 3: Scapy-radio architecture

It should be noted that, whenever an architectural decision was made, it was done keeping in mind that one day either GNU Radio or Scapy could be replaced by a new and more suitable component.

3.1 GNU Radio encapsulation

To be consistent with network encapsulation, Scapy relies on layer classes to understand and process network packets. Each layer will either be the payload of another layer or chained to multiple other ones.

To determine the first class layer to use, Scapy relies on the underlying network interface. However, in our case, this interface is going to send and receive different types of network packets depending on the wireless protocol GNU Radio is asked to work with.

To address this issue, we chose to add another layer on top of every network packet. It is a simple network header called *GnuradioPacket* and it is laid out in the following manner:

protocol	0x00	0x00	0x00	0x00	0x00	0x00	0x00

- 1 byte for protocol identification;
- 7 bytes reserved for future use (such as channel, RSSI or anything that would be packet-dependant).

In addition, we have already defined IDs for several wireless protocols:

Z-Wave	802.15.4	BT4LE	wM-Bus	DASH7
1	2	3	4	5

Here after is the corresponding Python code on Scapy's side:

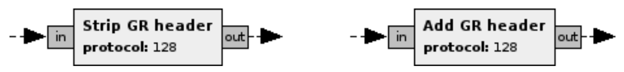
```
from scapy.layers.zwave import *
from scapy.layers.dot15d4 import *
from scapy.layers.bt4le import *
from scapy.layers.wmbus import *

class GnuradioPacket(Packet):
    name = "Gnuradio␣header"
    fields_desc = [
        ByteField("proto", 0),
        HiddenField(X3BytesField("reserved1", 0)),
        HiddenField(IntField("reserved2", 0))
    ]

bind_layers(GnuradioPacket, zwave, proto=1)
bind_layers(GnuradioPacket, dot15d4, proto=2)
bind_layers(GnuradioPacket, bt4le, proto=3)
bind_layers(GnuradioPacket, wmbus, proto=4)
#bind_layers(GnuradioPacket, dash7, proto=5)

#user-defined DLT for pcap files
conf.l2types.register(148, GnuradioPacket)
```

Since this encapsulation also needs to happen on GNU Radio's side, we have implemented two blocks that respectively strip and add our custom network header:



3.2 Scapy UDP SuperSocket

The process of sending packets in and out of Scapy is abstracted using a super-socket class. This makes the process of switching IO layers or even writing new ones very easy.

We did write a new one to establish bidirectional communications between GNU Radio and Scapy. It is called *GnuradioSocket* and it relies on two standard UDP sockets, one for input and one for output. Our choice was motivated by the following reasons:

- This does not require to be run as root (unlike TUN/TAP devices for example);
- Using UDP sockets in GNU Radio and Scapy is easy and it limits the amount of code that needs to be added on both ends;
- This may help us if we ever need to use multiple SDR boards at the same time (to tackle some of the limitations of SDR-based approaches).

To send packets in and out of Scapy, three main commands are natively supported: *sr* to send or receive multiple network packets, *sr1* to send or receive just one and *sniff* to gather every possible packets from the network interface. Therefore, based on *GnuradioSocket*, we have implemented their radio counterparts, *srradio*, *srradio1* and *sniffradio*:

```
@conf.commands.register
def srradio(pkts, inter=0.1, *args, **kargs):
    s = GnuradioSocket()
    a, b = sendrecv.sndrcv(s, pkts, inter=inter, *args, **kargs)
    s.close()
    return a, b

@conf.commands.register
def srradio1(pkts, *args, **kargs):
    a, b = srradio(pkts, *args, **kargs)
    if len(a) > 0:
        return a[0][1]

@conf.commands.register
def sniffradio(lsocket=None, radio=None, *args, **kargs):
    if radio is not None:
        switch_radio_protocol(radio)
    s = lsocket if lsocket is not None else GnuradioSocket()
    rv = sendrecv.sniff(lsocket=s, *args, **kargs)
    if lsocket is None:
        s.close()
    return rv
```

3.3 GNU Radio remote control

Scapy is fully scriptable and this is a valuable feature that we wanted to preserve with Scapy-radio. To achieve this, the first requirement is to be able to launch GNU Radio and execute a particular GRC flow graph from Scapy.

To that end, we have implemented a specific command with no counterpart in Scapy: *switch_radio_protocol*. Upon execution, this command will launch GNU Radio in a forked process in background and run the corresponding GRC flow graph after compiling it if this was not already done or if it was outdated. It should be noted that if this command is not called for the first time, everything will be cleaned-up before launching a new GNU radio process.

In addition, to do their work, GRC flow graphs may need several variables to be set (for example, an access code or a channel number). Just like any GNU Radio application, it can be done using the UI brought up by the execution of the GRC flow graph. However, to preserve scriptability, the second requirement is to be able to get and set these variables from Scapy.

To that end, we have implemented XMLRPC communications between Scapy and GNU Radio. For GRC flow graphs, we just added a native XMLRPC server block to each one of them. On Scapy's side, we added two new commands to abstract these communications, *gnuradio_get_vars* and *gnuradio_set_vars*:

```
@conf.commands.register
def gnuradio_get_vars(*args, **kargs):
    if "host" not in kargs:
        kargs["host"] = "127.0.0.1"
    if "port" not in kargs:
        kargs["port"] = 8080
    rv = {}
    try:
        import xmlrpclib
    except ImportError:
        print "xmlrpclib␣is␣missing␣to␣use␣this␣function."
    else:
        s = xmlrpclib.Server("http://%s:%d" %
                             (kargs["host"], kargs["port"]))
        for v in args:
            try:
                res = getattr(s, "get_%s" % v)()
                rv[v] = res
            except xmlrpclib.Fault:
                print "Unknown␣variable␣'%s'" % v
        s = None
    if len(args) == 1:
        return rv[args[0]]
    return rv

@conf.commands.register
def gnuradio_set_vars(host="localhost", port=8080, **kargs):
    try:
        import xmlrpclib
    except ImportError:
        print "xmlrpclib␣is␣missing␣to␣use␣this␣function."
    else:
        s = xmlrpclib.Server("http://%s:%d" % (host, port))
        for k, v in kargs.iteritems():
            try:
                getattr(s, "set_%s" % k)(v)
            except xmlrpclib.Fault:
                print "Unknown␣variable␣'%s'" % k
        s = None
```

4 Typical application: Z-Wave

Z-Wave is a proprietary wireless protocol developed by Zen-Sys, later acquired by Sigma Designs. It is designed for home automation, including access control and alarm systems.

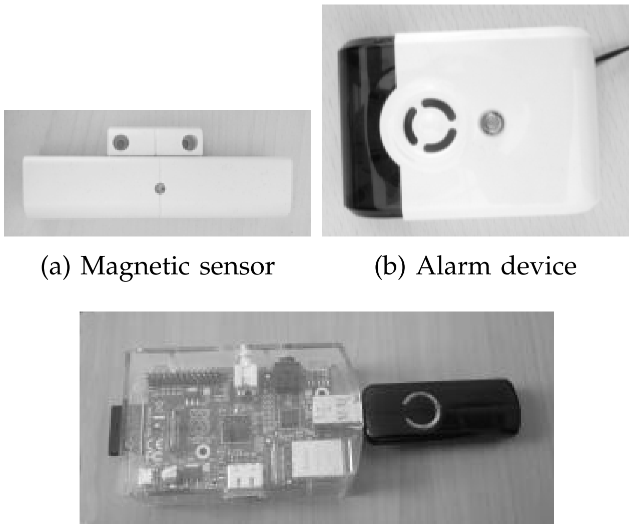
The security of this protocol was assessed for the first time last year [2]. It was relying on a sub-GHz radio transceiver paired with custom software and was successful at finding a vulnerability in the key exchange protocol.

Because of its recent worldwide adoption, its security-related applications and also because its very first assessment was able to find a critical vulnerability, Z-Wave was a perfect opportunity to put Scapy-radio to the test.

4.1 Test setup

To create a testing environment, we first designed a realistic scenario:

- A magnetic sensor (Figure 4a) is placed on a door so as to detect when someone successfully breaks in;
- When that happens, it sends a signal to a base station which, in return, sends a signal to turn an alarm device on (Figure 4b);
- When the rightful owner of the door comes in and gives the right security code to the base station, it sends a signal to turn the alarm device off.



(a) Magnetic sensor (b) Alarm device

(c) Raspberry Pi and network controller

Fig. 4: Z-Wave components

To act as a base station, a Raspberry Pi plugged with a Z-Wave USB network controller (Figure 4c) was programmed using OpenZWave development kit [11]. One known limitation of OpenZWave development kit is that it does not yet support encryption. However, this was not an issue since we did not plan on targeting the encryption layer.

4.2 Work and results

Based on various documents that could be found on the Internet, we quickly achieved a first implementation of the following three components:

- *gnuradio/grc/zwave.grc*: a GRC flow graph that can modulate and demodulate Z-Wave radio communications;
- *gnuradio/gr-zwave/**: a GNU Radio packet-sink block that can match and process Z-Wave packets at the PHY level;
- *scapy/layers/zwave.py*: a Scapy layer that can dissect and forge Z-Wave packets.

The goal now was to use Scapy-radio to verify our ability to listen for Z-Wave packets and refine our implementation. With this in mind, we ended up writing a script to discover Z-Wave devices and those they are communicating with. Hereafter is the script code:

```
if __name__ == "__main__":
  main.load_module('gnuradio')
  switch_radio_protocol("zwave")

  _seen = dict()
  if len(sys.argv) > 1:
    nbr_iter = int(sys.argv[1])
  else:
    nbr_iter = 20  # Default value

  try:
    while nbr_iter > 0:
      pkt = s.recv()
      if ZWaveReq in pkt:
        if pkt.homeid not in _seen:
          print "New␣home␣id␣:␣" + str(pkt.homeid)
          _seen[pkt.homeid] = dict()
        for dev in (pkt.src, pkt.dst):
          if dev not in _seen[pkt.homeid]:
            _seen[pkt.homeid][dev] = Zwave_device()
          if dev == pkt.dst:
            _seen[pkt.homeid][dev].type = \
              pkt[ZWaveReq].get_field('cmd').i2repr(pkt, pkt.cmd)
            _seen[pkt.homeid][dev].rec_from = pkt.src
          if dev == pkt.src:
            _seen[pkt.homeid][dev].send_to = pkt.dst
      nbr_iter -= 1
    display(_seen)

  except KeyboardInterrupt:
    display(_seen)
    sys.exit()
```

After multiple improvements of our Z-Wave implementation in Scapy-radio, we finally decided to attack our testing environment.

To that end, we listened for Z-Wave packets when the alarm device went on and when it went off. Thanks to the dissection capabilities of Scapy, it was easy to see that a simple *SWITCH_BINARY* (0x25) command was responsible for the change of state in both cases. The value sent with the command would accordingly be *ON* (0xFF) or *OFF* (0x00).

At that point in the analysis, to prevent the alarm device from ever going on, we only had to write an automaton that detects *SWITCH_BINARY_ON* commands and replay them as *SWITCH_BINARY_OFF*. Thanks again to Scapy capabilities, we achieved that in a very short time and, with this script running, no matter what we did with the magnetic sensor the alarm device never went on.

This result was expected, replaying packets when the communication channel is not encrypted is nothing new. The real achievement here is that we were able to carry out this attack, starting from square one, in a little less than a day. Hereafter is the automaton code:

```
from scapy.all import *

class Stop_alarm(Automaton):
    def parse_args(self, *args, **kargs):
        Automaton.parse_args(self, *args, **kargs)

    @ATMT.state(initial=1)
    def BEGIN(self):
        switch_radio_protocol("zwave")
        self.last_pkt = None
        print "BEGIN"
        raise self.WAITING()

    @ATMT.state()
    def WAITING(self):
        print "WAITING"

    @ATMT.receive_condition(WAITING)
    def alarm_on(self, packet_receive):
        human = lambda p,f:p.get_field(f).i2repr(p,getattr(p, f))
        if ZWaveReq in packet_receive:
            self.last_pkt = packet_receive
            if ZWaveSwitchBin in packet_receive:
                if human(packet_receive[ZWaveSwitchBin],
                         'switchcmd') == "SWITCH":
                    if human(packet_receive[ZWaveSwitchBin],
                             'val') == "ON":
                        raise self.WAITING()

    @ATMT.action(alarm_on)
    def alarm_off(self):
        time.sleep(0.5)
        print "TURNING␣ALARM␣OFF"
        pkt = self.last_pkt[ZWaveReq].copy()
        pkt[ZWaveSwitchBin].val = "OFF"
        pkt.seqn += 1
        pkt.crc = None
        self.send(pkt)

if __name__ == "__main__":
    load_module('gnuradio')
    Stop_alarm(debug=1).run()
```

5 LIMITED APPLICATION: BT4LE

Bluetooth 4.0 Low Energy (BT4LE) is a wireless area network technology designed for portable and wearable devices, including personal health and fitness applications.

This wireless protocol is famous for being difficult to handle for SDR-based approaches. Knowing that, we took that opportunity to confirm these limitations and to see if Scapy-radio could be any useful in tackling them.

5.1 Tested device

Figure 5 shows a health-related BT4LE device we could get our hands on: an e-cigarette. From the user manual that was supplied with this device, we noted three interesting points:

- It records user consumption;
- It uses a smartphone application;
- Over-the-air firmware update is possible.

The following issues may therefore be at stake:

- Privacy violation;
- Cascade-based attacks (compromise of a smartphone via the device);
- Over-the-air firmware corruption attacks and thus potential health issues.

Fig. 5: BT4LE e-cigarette

5.2 Work done

Based this time on full and easy-to-find specifications, we again quickly achieved a working implementation of the following components:

- *gnuradio/grc/bt4le.grc*: a GRC flow graph that can modulate and demodulate BT4LE radio communications;
- *gnuradio/gr-bt4le/**: a GNU Radio packet-sink block that can match and process BT4LE packets at the PHY level;
- *scapy/layers/bt4le.py*: a Scapy layer that can dissect and forge BT4LE packets.

BT4LE devices have two modes of operation: advertising and data. Advertising is the first step so this is where we started. This mode does not rely on channel hopping so listening for advertising packets was straightforward and there is nothing particular here to mention.

To establish a connection, we then forged and sent *CONNECT_REQ* packets. We knew that this time responses would be sent on data channels using a hopping sequence. To deal with this issue, we wanted to listen to multiple channels at the same time, thinking that with several trials we could capture some responses.

However, even in the event of a successful outcome, going any further would still be problematic: even with two SDR boards to work with enough channels at the same time, the mandatory response time for BT4LE data packets is 150µs. No matter what we did, we were never able to achieve this and that is where we stopped working for now.

5.3 Prospects

Because the information-processing chain will cause long response times, a channel-hopping sequence cannot be followed using SDR-based approaches. To try to circumvent this issue, one might want to listen to multiple channels at the same time which, if the radio spectrum is very large, may require multiple SDR boards.

Unfortunately, channel hopping also mandates short response times and for that there are no circumvention methods. In the end, this means the only way to solve this issue would be to use onboard FPGAs to follow channel-hopping sequences. This is not something we will consider in the future: it would break portability and take us back to the point where only specialists can improve our tool.

Nonetheless, there is still work to be done with BT4LE. For example, once a device has been identified, should it be through a SCAN_REQUEST/SCAN_RESPONSE dialog or through advertising packets, our tool can issue a CONNECT_REQUEST. It may be interesting to see how various devices handle bogus CONNECT_REQUEST packets.

In addition, this specific packet contains important parameters used to established the connection, such as:

- CHAN_MAP, a 40-bit field (1 bit per channel) that specifies the channels that can be used for communication;

- CRC_INIT, a seed for the CRC algorithm;
- CONNECTION_AA, an advertising address that will be used for connection and also as the access code that will allow us to match packets at the PHY level.

Using *CHAN_MAP*, it might be possible to force the device to use a given set of consecutive channels that we can listen with a single SDR board. It would then be easier to capture responses to bogus CONNECT_REQUEST packets for further analysis. We did not tried that because it would have required entirely reworking the GRC flowgraph to share a list of access addresses to look for at PHY level.

6 Conclusion

In this paper, we have presented Scapy-radio, a generic wireless monitor/injector tool designed for a wide range of wireless security assessments. By testing our tool on two use cases, we have demonstrated its effective penetration testing capabilities.

However, one use case also showed the limitations of SDR-based approaches when a radio communication system relies on channel hopping. This is known and this is why dedicated tools such as *Ubertooth* [4] exist.

Nonetheless, between the time a new wireless protocol reaches the market and the time a dedicated testing tool is released, we need to be able to carry out a first security assessment of the available devices.

In this paper, we have also demonstrated that our tool can effectively reduce the current complexity of carrying out such assessments.

7 Code release

Scapy-radio is free, open and due to be mainstreamed in Scapy. In the meanwhile, get it at *http://bitbucket.cassidiancybersecurity.com*. Here is what we are releasing with this paper:

- A modified version of Scapy that includes the following layers:
  - 802.15.4, XBee, ZigBee, 6LoWPAN (last two forked from Scapy-com);
  - Z-Wave;
  - BT4LE;
  - wM-Bus.
- GRC flow graphs (that rely on UHD):
  - 802.15.4;
  - Z-Wave;
  - BT4LE (advertising only).

It should be mentioned that support for protocols other than Z-Wave and BT4LE advertising have not been thoroughly tested. Do not expected them to be fully functional.

8 Acknowledgement

This work was funded by the cybersecurity division of AIRBUS Defence & Space.

References

[1] DRAFT NISTIR 7628 Revision 1, Guidelines for Smart Grid Cyber Security, volume 3, Page 85.
[2] Behrang Fouladi and Sahand Ghanoun, Security Evaluation of the Z-Wave Wireless Protocol, Black Hat USA 2013.
[3] http://code.google.com/p/rfcat
[4] http://ubertooth.sourceforge.net
[5] http://riverloopsecurity.com/projects.html
[6] http://greatscottgadgets.com/hackrf
[7] http://nuand.com
[8] http://ettus.com/product/category/USRP-Bus-Series
[9] http://gnuradio.org/redmine/projects/gnuradio/wiki
[10] http://secdev.org/projects/scapy
[11] http://openzwave.com

Jean-Michel Picod is currently working at AIRBUS Defence & Space as the technical leader for incident response, forensic analysis and reverse engineering activities. He holds a master's degree in computer engineering, has contributed on several open source projects and published several open source tools such as DPAPIck, OWADE and various forensic scripts.

Arnaud Lebrun is an electronics and automation engineer currently working on wireless communications and industrial network security at AIRBUS Defence & Space. He holds a master's degree in electrical and electronic engineering and has been conducting vulnerability research activities on industrial control systems, private mobile radio networks and smart grids.

Jonathan-Christofer Demay, PhD is an IT security specialist with diverse professional backgrounds. As an academic researcher, he has been working on IDS bypassing, intrusion detection and general network security. As a consultant for various strategic industries and government bodies, he has been working on incident response, penetration testing and social engineering.