Global Information Assurance Certification Paper

A Black-Box Approach to Embedded Systems Vulnerability Assessment

GIAC GSEC Gold Certification

Author: Michael Horkan, mhorkan4223@gmail.com
Advisor: Chris Walker
Accepted: November 28, 2016

Abstract

Vulnerability assessment of embedded systems is becoming more important due to security needs of the ICS/SCADA environment as well as the emergence of the Internet of Things (IoT). Often, these assessments are left to test engineers without intimate knowledge of the device's design, no access to firmware source or tools to debug the device while testing. This gold paper will describe a test lab black-box approach to evaluating an embedded device's security profile and possible vulnerabilities. Open-source tools such as Burp Suite and python scripts based on the Sulley Fuzzing Framework will be employed and described. The health status of the device under test will be monitored remotely over a network connection. I include a discussion of an IoT test platform, implemented for Raspberry Pi, and how to approach the evaluation of IoT using this device as an example.

1. Introduction

With a continually growing emphasis on the security of ICS (Industrial Control Systems), SCADA (Supervisory Control And Data Acquisition), and IoT (Internet of Things) comes a developing interest in the vulnerability assessment of embedded devices. Standard security test techniques such as network and file fuzzing, protocol analysis, and vulnerability scanning are viable approaches. What makes embedded devices a unique challenge as compared to Linux- or Windows- based servers is their specialized, often custom, real-time operating systems, and the limited interface options that they expose. Debugging running firmware on these devices often requires specialized tools - both hardware and software. There are industry-standard recommendations detailing the need for hacking embedded device hardware when possible (Searle). These tools are often not available to test engineers due to availability and cost.

Hardware hacking may not be an option for a variety of reasons. JTAG (Joint Test Action Group) interfaces, used by manufacturers for development and debug purposes, may be secured or disabled. On-board firmware may also be encrypted, preventing dumping and analysis. (Rippel, 2016)

For these reasons, test engineers often face the task of evaluating the security profile of embedded devices while not having access to the software under the hood. This leaves test engineers with the task of evaluating the security of embedded systems using only the interfaces that the device presents to users. They lack the benefits of developer access, source code, design documentation, and specialized debug tools. This is called “black box” testing (the inverse is “white box” or “crystal box”).

In the following sections, I will demonstrate test strategies for uncovering security vulnerabilities in embedded systems when disassembly and data dumping are impractical options. I will use a business-class Ethernet switch as a test target for these strategies. It is important for automated testing to account for the “health” of the device during test cases. Without the ability to install a debugger in the device’s operating system, testers must use alternative methods of evaluating device health. These methods will focus on the target’s ability to continue to provide essential services, while network packets are directed to the target with improper payloads.

Michael Horkan,
mhorkan4223@gmail.com

Testing for network response is not the ideal way to determine that a process has failed. However, if a tester can isolate a test case that results in a denial-of-service condition, supplying a proof-of-concept script and a packet capture of the test case to a firmware developer is an effective way of addressing a potential vulnerability.

DISCLAIMER: The techniques described in this paper should never be attempted against systems running in production. They could result in damage to equipment, production resources, the environment, or human life depending upon the application of the control system. They should be performed only against embedded systems in a test bed staging environment.

2. The Test Target

2.1. Test Setup

In the following example I provide, I use a Cisco Catalyst 2950 managed Ethernet switch as the target for security evaluation. The test setup diagram follows:

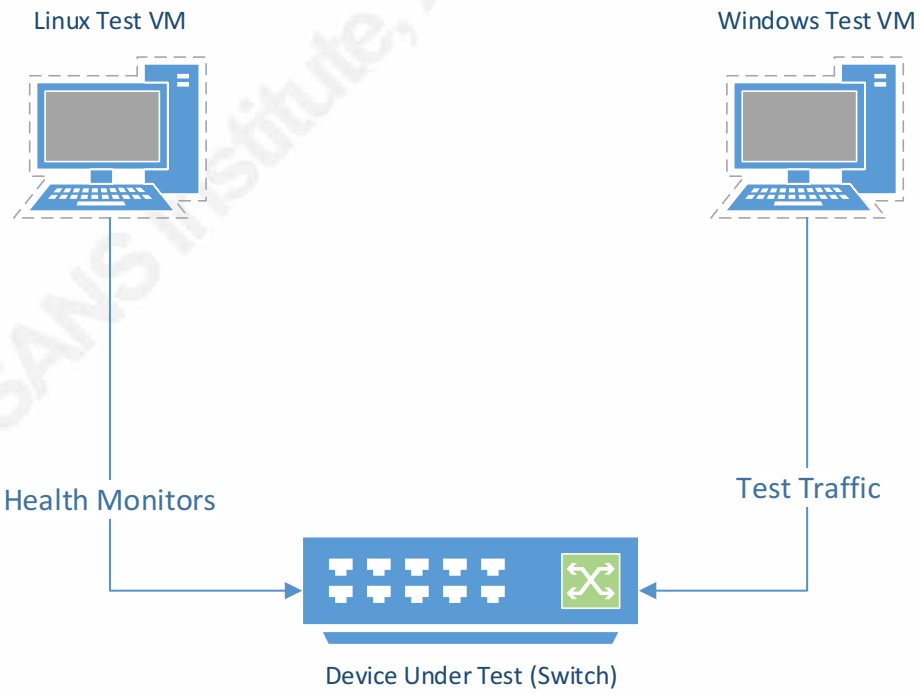

Figure 1 Test Setup

Michael Horkan,
mhorkan4223@gmail.com

2.2. Device Profile

The switch supports the following interfaces:

- Telnet (TCP port 23)
- HTTP (Hyper-Text Transfer Protocol, TCP port 80)
- SNMP (Simple Network Management Protocol, UDP port 161)

So far in the evaluation, the use of Telnet can be flagged as an issue that future releases of the product's firmware must resolve.

Figure 2 shows the device's home web page.

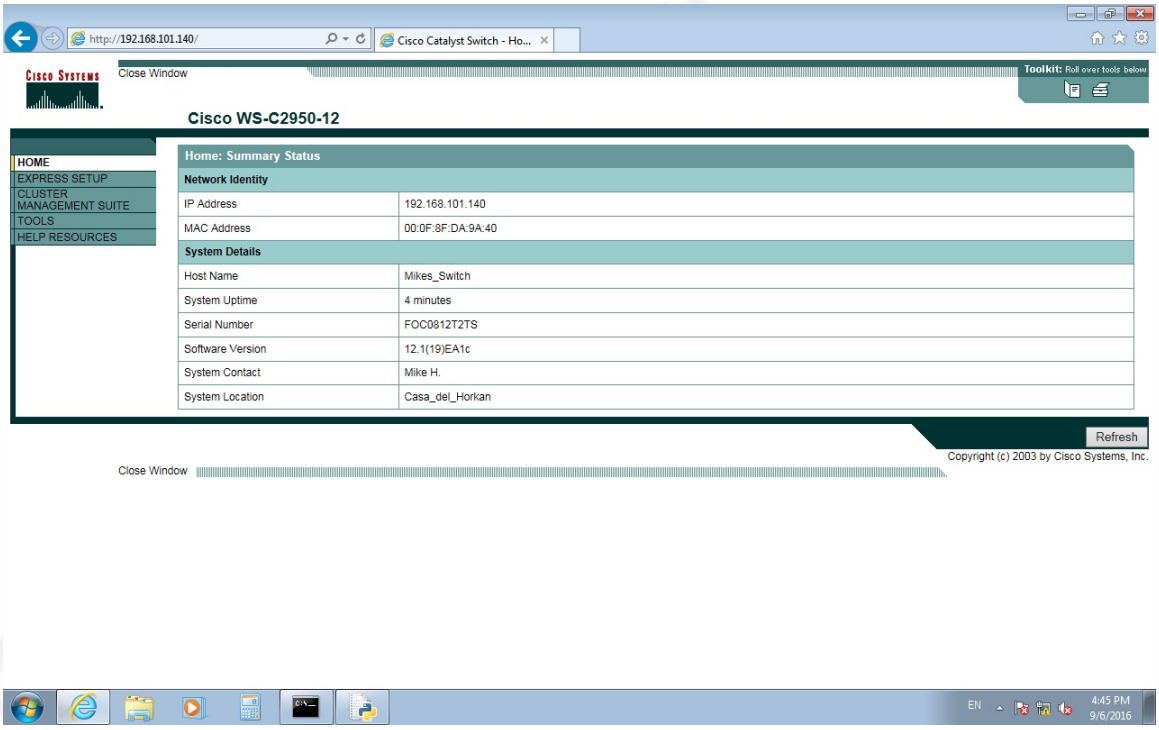

Figure 2 Device Under Test Home Web Page

Michael Horkan,
mhorkan4223@gmail.com

3. Vulnerability Assessment Strategies

3.1. Protocol and Packet Analysis

A network-based vulnerability analysis begins with a simple examination of the protocols that the device uses under normal operating conditions.

The use of insecure, plain-text protocols such as Telnet, SNMP (versions 1 and 2c), and FTP is a problem. Security best practices advise against their use. An attacker performing a simple snoop of the network traffic – either using a data tap, a span port on a switch, or a man-in-the-middle attack – can extract unencrypted credentials from these protocols.

Figure 3 shows a Wireshark display of a packet capture of an SNMP read to the switch. The community string is plainly readable.

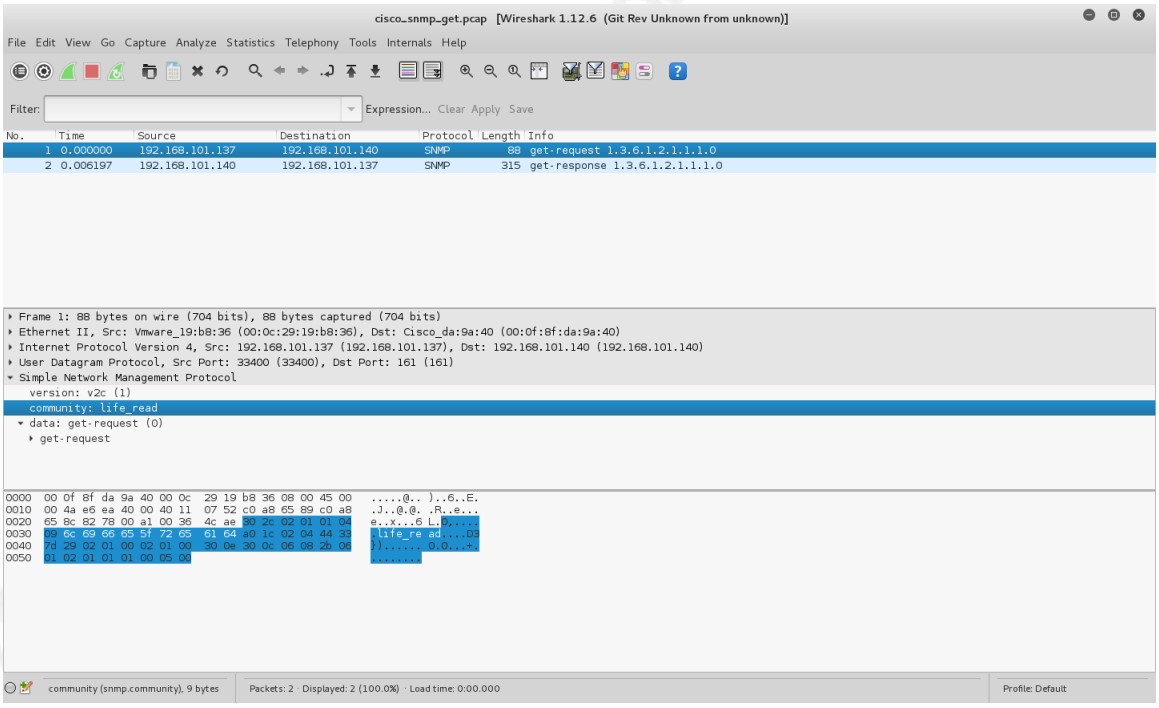

Figure 3 Packet Capture of SNMP Read showing plain-text community string

Many embedded devices employ custom web servers as configuration interfaces. To analyze HTTP/HTTPS traffic to and from such devices, Portswigger's Burp Suite and OWASP's Zap are excellent choices as web proxies. Both products are fully-featured web proxies, capable of decoding traffic, replacing packet fields, blocking or

Michael Horkan,
mhorkan4223@gmail.com

transparently logging web traffic. Zap is free. Burp Suite comes in a free version and in a rather inexpensive (roughly three-hundred US dollars) professional edition.

Setting up Burp Suite on the Windows Test VM, a tester can capture HTTP requests to the switch's web server. Figure 4 shows the result of a proxy history during login to the web server. The HTTP Basic Authentication string is in Base64 format as "0mxpZmVqYWNrZXQ=". This string decodes to plain text as "lifejacket".

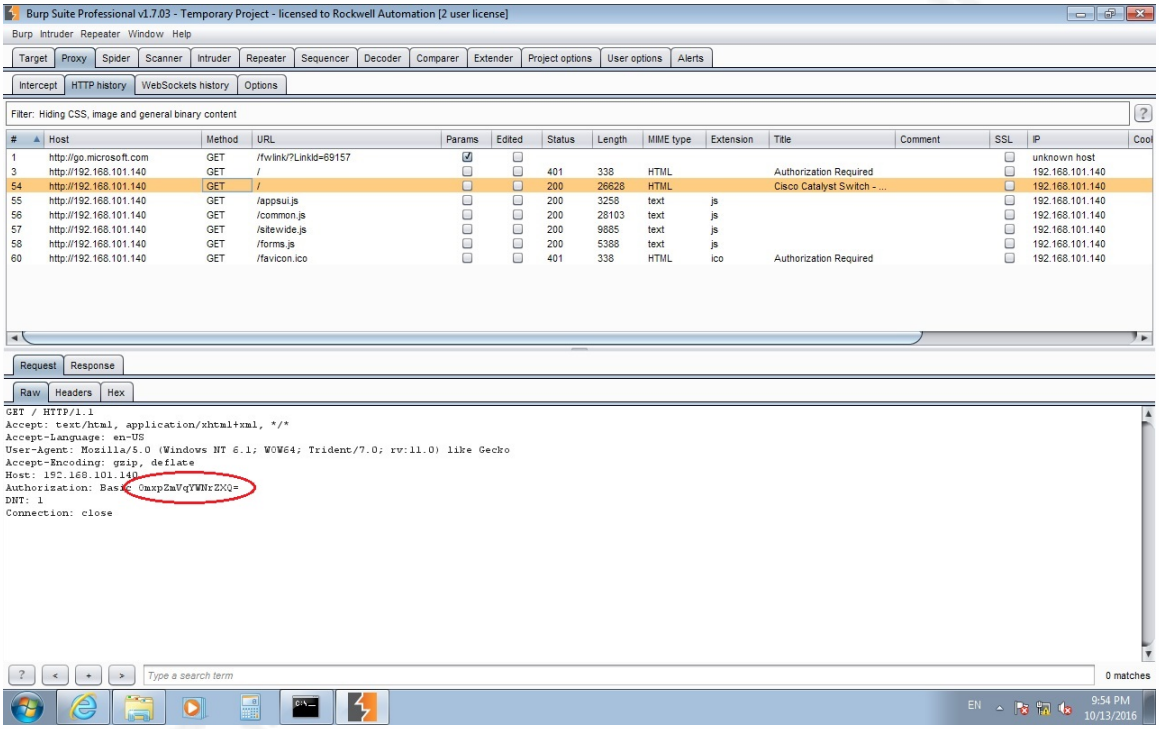

Figure 4 Burp Proxy History Showing Web Server Login

3.2. Network-Based Fuzz Testing

Fuzz testing is the practice of delivering deliberately improper, often randomized, input to a software application in order to determine if the application is vulnerable to buffer overflow errors or other undesirable effects. Many network-based software exploits take advantage of buffer overflows to compromise the security of software servers and embedded devices that consume malformed packets. Network packets that an application consumes can be injected with bad data – "fuzzed" – and the target application monitored to determine if correct operation has ceased. We can use fuzzing tools customized to the three identified network protocols to check for buffer overflow errors in the switch's firmware. Standard network fuzzing tools such as Spike, Sulley

Michael Horkan,
mhorkan4223@gmail.com

Fuzzing Framework, and sfuzz are good choices for these test cases. These tools have extensive online documentation available, and are customizable and free.

To begin, I will use Sulley Fuzzing Framework to mutate an HTTP GET method request to the switch's home page. The authenticated request in HTTP plain text is as follows:

```
GET / HTTP/1.1
Accept: text/html, application/xhtml+xml, */*
Accept-Language: en-US
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64;
Trident/7.0; rv:11.0) like Gecko
Accept-Encoding: gzip, deflate
Host: 192.168.101.140
Authorization: Basic OmxpZmVqYWNrZXQ=
DNT: 1
Connection: close
```

I created a Sulley Python script to fuzz every text field individually, leaving delimiters such as spaces, carriage returns, colons, and slashes intact. See Appendix A for the script listing "http_GET_sulley_session.py".

The second fuzzed network packet was modeled using the Linux Net-SNMP toolset "snmpset" command. The command line for setting the device's Location (OID iso.3.6.1.2.1.1.6.0) is:

```
snmpset –v2c –c life_write 192.168.101.140 1.3.6.2.1.1.6.0 s
                    "Casa_del_Horkan_2"
```

The SET command was modeled using Sulley. See the code listed in Appendix A as "snmp_write_sulley_session.py".

The test is being employed to check for conditions which will cause the device under test to cease functioning as it should. A buffer overflow will almost always cause a software crash. In order to detect this condition, it is necessary to periodically probe the device's essential services for signs of failure. I refer to the scripts that check these services as "health monitors".

Michael Horkan,
mhorkan4223@gmail.com

3.3. Health Monitors

During fuzz testing, it's necessary to verify the health of the device by continually testing the availability of essential services. For this, scripts will be employed that do the following:

HTTP Monitor – Periodic GET to root ("/")

SNMP Monitor – Periodic SNMP Get of the System Description parameter

Telnet Monitor – Periodic login with a valid password

Connection Monitor – Ubuntu client is periodically checked with ICMP Echo request, proving that the switch is still passing network traffic.

The general attribute shared by these monitors is that they will continuously test for service at user-configurable time periods. They will scroll status – success and failure – and not terminate based on this status. Timestamps will be printed with the statuses so that a user can easily see that the monitor is still functioning, and correlate changes in status with test conditions.

Each monitor will print an identification string with every status line output, to make it easier for the user to distinguish between monitors running in parallel and in separate terminal sessions.

See Appendix A for a listing of the monitor Python scripts.

3.3.1. HTTP Monitor

See Appendix A for a listing of "http_monitor.py". The command-line for this script's use is:

```
./http_monitor.py <IP address of target> <period in seconds>
```

This monitor will connect to port 80 on the target device and submit an HTTP GET request for the root directory "/". As long as the target's web server is listening on port 80, and the server process is still operational, the monitor should receive an HTTP response. A valid HTTP response means the server is still operational. Most likely the response will be HTTP 401 (Unauthorized) if the web server demands authentication,

Michael Horkan,
mhorkan4223@gmail.com

HTTP 302 (Redirect Found) if the web server redirects to a different document path or another port (such as TCP/443).

Figure 5 shows the monitor's output when the web server is up and listening. The monitor prints the HTTP response code it receives from the server, in the example below the response is 401:

```
root@sad-sack:~/Documents/python_monitors# ./http_monitor.py 192.168.101.140 2
[HTTP Monitor] Tue Aug 16 18:17:10 2016 401
[HTTP Monitor] Tue Aug 16 18:17:12 2016 401
[HTTP Monitor] Tue Aug 16 18:17:14 2016 401
[HTTP Monitor] Tue Aug 16 18:17:16 2016 401
```

Figure 5 HTTP Monitor showing successful GETs

The monitor will display a failure, printing "timeout", if the connection to port 80 fails, or the web server fails to return an HTTP response. Figure 6 below shows a service failure.

```
[HTTP Monitor] Tue Aug 16 18:18:33 2016 timeout
[HTTP Monitor] Tue Aug 16 18:18:40 2016 timeout
[HTTP Monitor] Tue Aug 16 18:18:47 2016 timeout
[HTTP Monitor] Tue Aug 16 18:18:54 2016 timeout
[HTTP Monitor] Tue Aug 16 18:19:01 2016 timeout
```

Figure 6 HTTP Monitor showing timeout failures

3.3.2. SNMP Monitor

See Appendix A for a listing of "snmp_monitor.py". The command-line for this script's use is:

```
./snmp_monitor.py <IP address of target> <community string>
<period in seconds>
```

This script performs a SNMP Get request for the System Description (OID 1.3.6.1.2.1.1.1.0), a SNMP MIB entry that should be present in all SNMP-capable devices. It requires the use of a community string that satisfies at least Read-Only level access. Under the hood, the monitor requires the use of the PySNMP library (pysnmp.sourceforge.net).

Figure 7 shows the monitor's output under normal operating conditions. The monitor receives and prints the system description that the device responds with.

Michael Horkan,
mhorkan4223@gmail.com

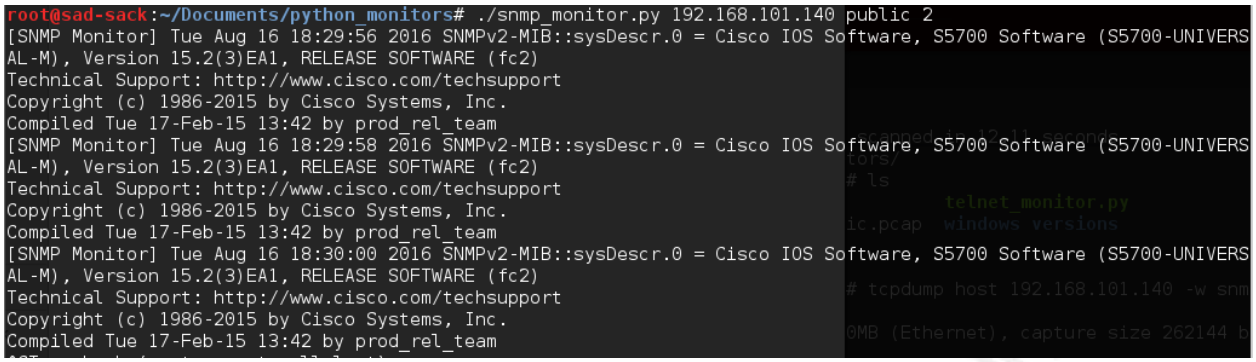

Figure 7 SNMP Monitor showing successful operation

Figure 8 shows service failure. If the device does not respond successfully, the monitor will print the PySNMP "RequestTimedOut" error:

Figure 8 SNMP Monitor failure

Michael Horkan,
mhorkan4223@gmail.com

3.3.3. Telnet Monitor

See Appendix A for a listing of "telnet_monitor.py". The command-line for this script's use is:

```
./telnet_monitor.py <IP address of target> <telnet password>
<period in seconds>
```

This monitor verifies that the device under test's telnet server is operational. The script takes advantage of python's built-in "telnetlib" library. The monitor attempts to log in to the telnet server and reports success if it detects the telnet shell prompt.

Figure 9 shows the monitor's output under normal operating conditions, printing "Login Successful!" if a telnet shell prompt is received:

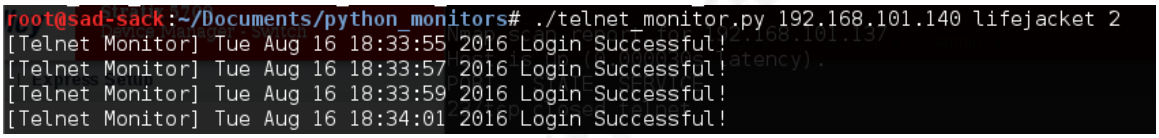

Figure 9 Telnet Monitor showing normal operating conditions

Figure 10 shows service failure, the result of a timeout.

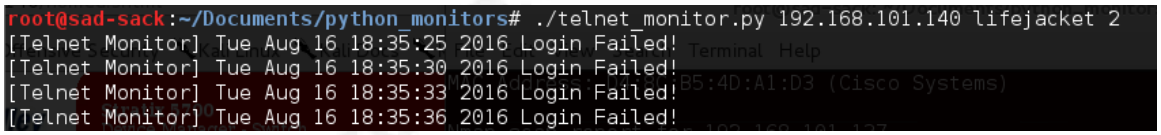

Figure 10 Telnet Monitor Failure Status

3.3.4. Ping Monitor

See Appendix A for a listing of "ping_monitor.py". The command-line for this script's use is:

```
./ping_monitor.py <IP address of target> <period in seconds>
```

This monitor sends an ICMP Echo Request to the IP address and reports success if a response is received. An ICMP message cannot be sent using Python's socket object because it operates at the Transport layer (layer 4) of the OSI model. ICMP operates at the Network Layer (layer 3), so Scapy was used to formulate and send the packet.

Figure 11 shows the monitor's output under normal operating conditions.

Michael Horkan,
mhorkan4223@gmail.com

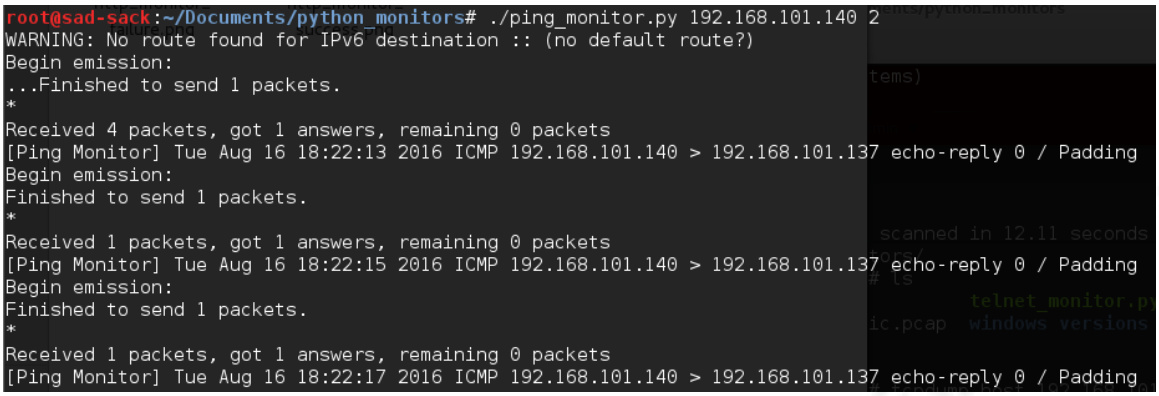

Figure 11 Ping Monitor Output under Normal Operating Conditions

Figure 12 shows ping monitor failure.

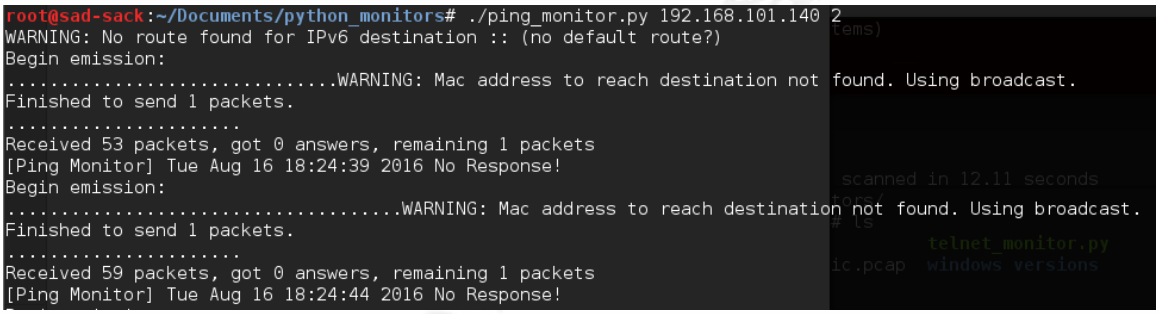

Figure 12 Ping Monitor Failure Status

4. Conclusions and Further Considerations

It is important for vendors of ICS/SCADA/IoT to test their products for security vulnerabilities, to protect their customers from attacks. This paper has described an inexpensive and valid approach to assessing the security profile of an embedded system in a test environment using black-box network-based techniques. It provided an approach for engineers who are less experienced in security testing to evaluate an embedded system, without the need to purchase expensive software licenses or test equipment. There are further steps to allow a purely automated approach to the fuzz testing portion of the evaluation.

The ability to automatically log a test case as being responsible for a service failure is necessary for test harness automation. The scripts I have supplied do not provide this functionality. The script supplying test cases (mutated packets) would need to communicate across the network to the health monitors in order for this function to occur.

Michael Horkan,
mhorkan4223@gmail.com

Once a health monitor detects a failure, it must log the current test case for failure attribution. A Python-based network communication library such as Twisted may be able to provide the necessary functions (https://twistedmatrix.com/trac/).

For an automated fuzzing test, the test harness should be capable of resetting a device that has lost service capability due to a firmware crash. It is possible to cycle power to an embedded system and return it to an operational state, provided that the test case did not result in stored firmware sustained damage. One simple way to perform this function is through a USB-connected input/output device that can be controlled using scripts. Devices such as those found at https://labjack.com provide an easy and inexpensive way to reset devices under test. Figure 13 shows how this device connects to a test workstation in order to control 120 VAC power for a device under test.

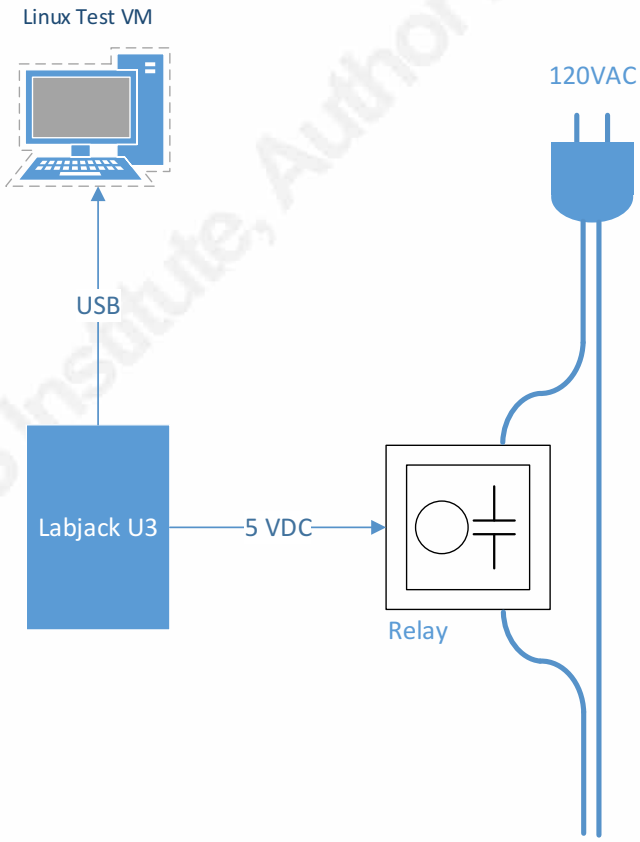

Figure 13 Connection Diagram for Resetting Device Under Test Power

Automated testing is only part of the picture in security assessment. Manual firmware analysis is another avenue for device assessment. For firmware that is

Michael Horkan,
mhorkan4223@gmail.com

unencrypted, tools like the Linux 'strings' command can be used to reveal default passwords and the presence of familiar software libraries that may uncover known vulnerabilities. The open-source 'binwalk' tool (binwalk.org) can be used to detect and extract firmware component files for disassembly. This type of analysis falls more squarely in the realm of penetration testing, being more "hands-on" and dependent on tester experience for success, and therefore not a candidate for automated testing.

Michael Horkan,
mhorkan4223@gmail.com

5. Appendix A – Python Script Listings

http_monitor.py

```
#!/usr/bin/env python

import httplib
import sys
import time

def send_http_req(ip, url):
    try:
        conn = httplib.HTTPConnection(ip, 80, timeout=5)
        conn.request("GET", url)
        r1 = conn.getresponse()
    except:
      return "timeout"

    conn.close()
    return r1.status

if __name__ == "__main__":

    if len(sys.argv) < 3:
      print("Usage: http_monitor.py <ip address of target> <test
      period in seconds>")
      sys.exit()

    ip = sys.argv[1]
    period = int(sys.argv[2])

    # request root from web server
    url_requested = "/"

    while True:
        return_status = send_http_req(ip, url_requested)
     printable_time = time.asctime(time.localtime(time.time()))
        print("[HTTP Monitor] " + printable_time + " " +
        str(return_status))
        time.sleep(period)
```

Michael Horkan,
mhorkan4223@gmail.com

snmp_monitor.py

```
#!/usr/bin/env python

from pysnmp.hlapi import *
import sys
import time

def send_snmp_get(ip, community_string):

    # Request OID 1.3.6.1.2.1.1.1.0, which is the System
Description

    errorIndication, errorStatus, errorIndex, varBinds = next(
        getCmd(SnmpEngine(),
               CommunityData(community_string, mpModel=0),
               UdpTransportTarget((ip, 161)),
               ContextData(),
               ObjectType(ObjectIdentity("1.3.6.1.2.1.1.1.0")))
    )

    printable_time = time.asctime(time.localtime(time.time()))

    if errorIndication:
      print("[SNMP Monitor] " + printable_time + " " +
      repr(errorIndication))
    elif errorStatus:
      print("[SNMP Monitor] " + printable_time + " " + '%s at %s'
      % (errorStatus.prettyPrint(),
       errorIndex and varBinds[int(errorIndex) - 1][0] or "?"))
    else:
        for varBind in varBinds:
           print("[SNMP Monitor] " + printable_time + " " + " =
           ".join([x.prettyPrint() for x in varBind]))

if __name__ == "__main__":

    if len(sys.argv) < 4:
      print("Usage: snmp_monitor.py <ip address of target>
      <community string> <test period in seconds>")
      sys.exit()

    ip = sys.argv[1]
    community_string = sys.argv[2]
    period = int(sys.argv[3])

    while True:
        send_snmp_get(ip, community_string)
        time.sleep(period)
```

Michael Horkan,
mhorkan4223@gmail.com

ping_monitor.py

```
#!/usr/bin/env python

from scapy.all import sr1, IP, ICMP
import sys
import time

def send_icmp_echo(ip):

    packet = sr1(IP(dst=ip)/ICMP(), timeout=1)
    printable_time = time.asctime(time.localtime(time.time()))

    if packet:
      print("[Ping Monitor] " + printable_time + " " +
      packet[0][ICMP].summary())
    else:
      print("[Ping Monitor] " + printable_time + " No Response!")

if __name__ == "__main__":

    if len(sys.argv) < 3:
      print("Usage: ping_monitor.py <ip address of target> <test
      period in seconds>")
      sys.exit()

    ip = sys.argv[1]
    period = int(sys.argv[2])

    while True:
        send_icmp_echo(ip)
        time.sleep(period)
```

Michael Horkan,
mhorkan4223@gmail.com

telnet_monitor.py

```
#!/usr/bin/env python

import telnetlib
import sys
import time

def telnet_login(ip, pw):

    try:
        tn = telnetlib.Telnet(ip)

        pw_prompt_string = tn.read_until("Password: ", timeout=2)
        if "Password: " in pw_prompt_string:
            tn.write(pw + "\n")
        else:
            raise Exception("Password Prompt Not Received!")

        prompt_string = tn.read_until("Mikes_Switch>", timeout=2)
        if "Mikes_Switch>" in prompt_string:
            pass
        else:
            raise Exception("Telnet Prompt Not Received!")

        tn.close()
       printable_time =
     time.asctime(time.localtime(time.time()))
        print("[Telnet Monitor] " + printable_time + " Login
        Successful!")

    except:
      printable_time = time.asctime(time.localtime(time.time()))
      print("[Telnet Monitor] " + printable_time + " Login
      Failed!")


if __name__ == "__main__":

    if len(sys.argv) < 4:
      print("Usage: telnet_monitor.py <ip address of target>
      <password> <test period in seconds>")
      sys.exit()

    ip = sys.argv[1]
    pw = sys.argv[2]
    period = int(sys.argv[3])

    while True:
        telnet_login(ip, pw)
        time.sleep(period)
```

Michael Horkan,
mhorkan4223@gmail.com

http_GET_sulley_session.py

```
from sulley import *
import sys
import time

s_initialize("HTTP_GET")
s_static("GET")
s_delim(" ")
s_string("/")
s_delim(" ")
s_string("HTTP")
s_delim("/")
s_string("1.1")
s_delim("\r\n")
s_string("Accept")
s_delim(":")
s_delim(" ")
s_string("text")
s_delim("/")
s_string("html")
s_delim(",")
s_delim(" ")
s_string("application")
s_delim("/")
s_string("xhtml")
s_delim("+")
s_string("xml")
s_delim(",")
s_delim(" ")
s_string("*")
s_delim("/")
s_string("*")
s_delim("\r\n")
s_string("Accept-Language")
s_delim(":")
s_delim(" ")
s_string("en-US")
s_delim("\r\n")
s_string("User-Agent")
s_delim(":")
s_delim(" ")
s_string("Mozilla")
s_delim("/")
s_string("5.0")
s_delim(" ")
s_delim("(")
s_string("Windows")
s_delim(" ")
s_string("NT")
```

Michael Horkan,
mhorkan4223@gmail.com

```
s_delim(" ")
s_string("6.1")
s_delim(";")
s_delim(" ")
s_string("WOW64")
s_delim(";")
s_delim(" ")
s_string("Trident")
s_delim("/")
s_string("7.0")
s_delim(";")
s_delim(" ")
s_string("rv")
s_delim(":")
s_string("11.0")
s_delim(")")
s_delim(" ")
s_string("like")
s_delim(" ")
s_string("Gecko")
s_delim("\r\n")
s_string("Accept-Encoding")
s_delim(":")
s_delim(" ")
s_string("gzip")
s_delim(",")
s_delim(" ")
s_string("deflate")
s_delim("\r\n")
s_string("Host")
s_delim(":")
s_delim(" ")
s_string("192.168.101.140")
s_delim("\r\n")
s_string("Authorization")
s_delim(":")
s_delim(" ")
s_string("Basic")
s_delim(" ")
s_string("OmxpZmVqYWNrZXQ=")
s_delim("\r\n")
s_string("DNT")
s_delim(":")
s_delim(" ")
s_string("1")
s_delim("\r\n")
s_string("Connection")
s_delim(":")
s_delim(" ")
s_string("close")
s_delim("\r\n")
s_delim("\r\n")
```

Michael Horkan,
mhorkan4223@gmail.com

```
print "Mutations: " + str(s_num_mutations())

print "Press CTRL/C to cancel in ",
for i in range(3):
      print str(3 - i) + " ",
      sys.stdout.flush()
      time.sleep(1)

print "Instantiating session"
sess = sessions.session(session_filename="http_get.session",
sleep_time=0.25)

print "Instantiating target"
target = sessions.target("192.168.101.140", 80)

print "Adding target"
sess.add_target(target)

print "Building graph"
sess.connect(s_get("HTTP_GET"))

print "Starting fuzzing now"
sess.fuzz()
```


Michael Horkan,
mhorkan4223@gmail.com

snmp_write_sulley_session.py

```
from sulley import *
import sys
import time

s_initialize("SNMP_WRITE")
s_binary("0x30 0x3e 0x02 0x01") # header
s_byte("\x01") #version 2c (1)
s_binary("0x04 0x0a") # ???
s_string("life_write") # community string
s_binary("0xa3 0x2d 0x02 0x04") # set-request header
s_binary("0x0f 0xf9 0x8f 0x19") # request-id 268013337, Big
Endian
s_binary("0x02 0x01") # ???
s_byte("\x00") # error-index 0
s_binary("\\x30\\x1f") # ???
s_binary("\\x30\\x1d") # variable-bindings 1 item
s_binary("\\x06\\x08") # Object Name
s_binary("\\x2b\\x06\\x01\\x02\\x01\\x01\\x06\\x00") #
iso.3.6.1.2.1.1.6.0 Device Location
s_binary("\\x04\\x11") # Value (OctetString)
s_string("Casa_del_Horkan_2")


print("Mutations: " + str(s_num_mutations()))

print("Press CTRL/C to cancel in "),
for i in range(3):
     print(str(3 - i) + " "),
     sys.stdout.flush()
     time.sleep(1)

print("Instantiating session")
sess = sessions.session(session_filename="snmp_write.session",
sleep_time=0.25, proto="UDP")

print("Instantiating target")
target = sessions.target("192.168.101.140", 161)

print("Adding target")
sess.add_target(target)

print("Building graph")
sess.connect(s_get("SNMP_WRITE"))

print("Starting fuzzing now")
sess.fuzz()
```

Michael Horkan,
mhorkan4223@gmail.com

6. Appendix B – Analysis of Internet-of-Things (IoT) Simulator

The explosion of Internet-of-Things devices in the marketplace makes their study and testing particularly interesting. They are in fact embedded systems, similar to those employed in industrial applications. For this reason, a discussion of their security profile is appropriate.

For testing purposes, an engineer can use a Raspberry Pi single-board computer as an Internet-of-Things (IoT) simulator. There is a recipe available online for connecting a Raspberry Pi to IBM's Watson IoT platform. (IBM Developer Recipes, 2015). It describes installing a new service on the Pi for communication to IBM's developer cloud. Figure 14 shows the architecture for this setup.

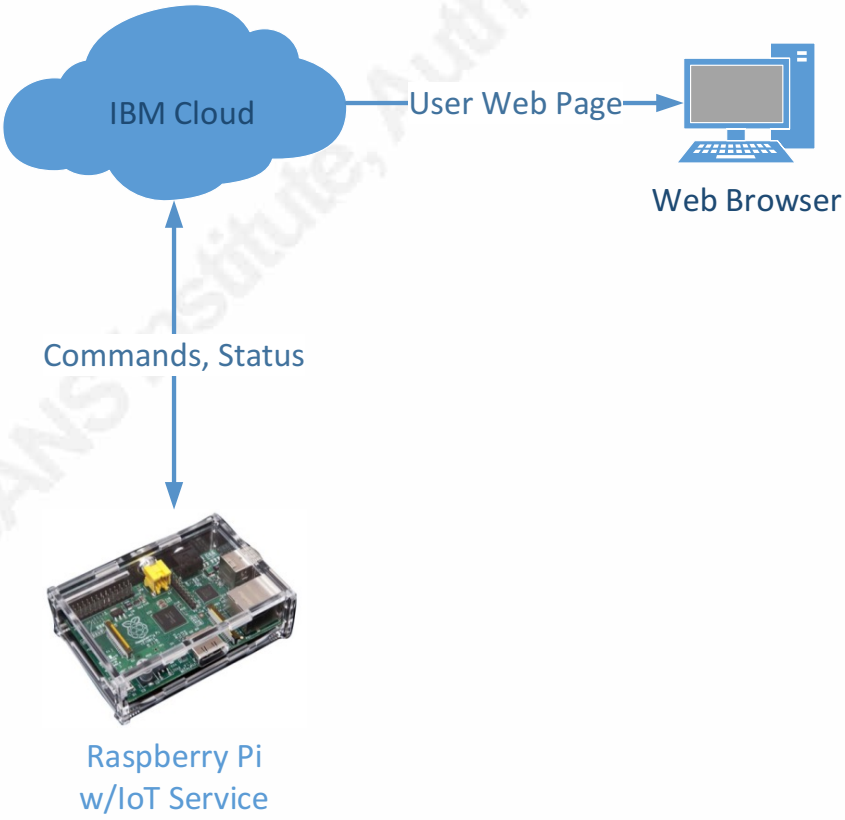

Figure 14 Raspberry Pi Setup with IBM Cloud Service

Michael Horkan,
mhorkan4223@gmail.com

By default, the IoT service generates data for transmittal to the cloud service such as CPU Utilization, CPU Temperature, and Memory Usage. IBM Watson graphs the data on a web page searchable by Device ID, and shown in Figure 15.

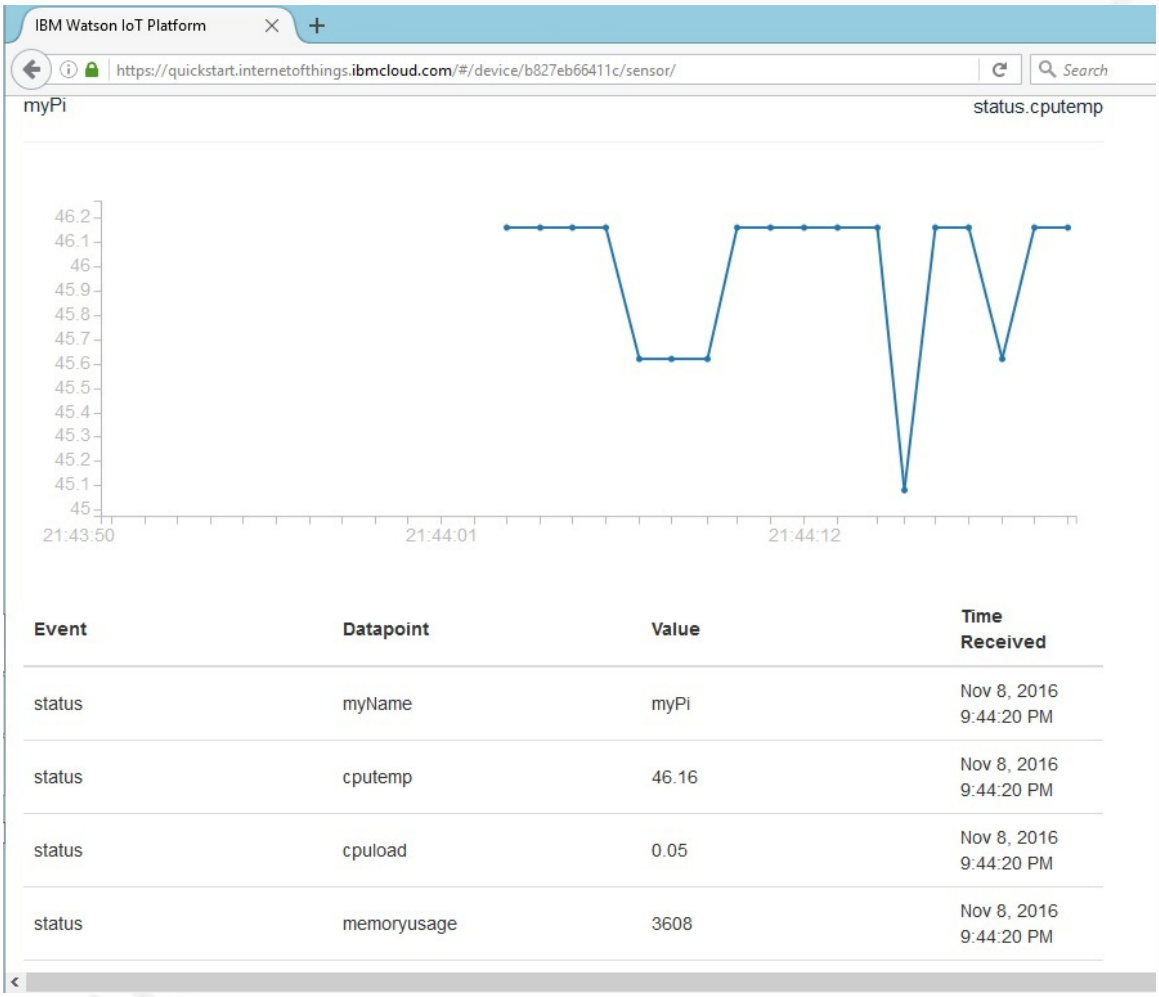

Figure 15 IBM Watson Raspberry Pi Device Page

Treating the Pi as a generic IoT target for assessment, nmap shows only port 22/tcp (SSH) listening; this is generic to the Pi's Rasbian distribution, and indicates that the IoT service is a client to IBM's cloud platform. This means that our previously-described server-centric fuzzing techniques will not work to evaluate the target in this case. Instead, a test-bench cloud server would have to be created to fuzz responses to the client. Were this a cloud application that sent commands to the IoT device to change I/O states or modify the configuration, such messages would make ideal targets.

Michael Horkan,
mhorkan4223@gmail.com

The Pi communicates to the home router wirelessly, as many IoT devices do. Protocol analysis of the traffic between the Pi and the cloud begins with setting up a wireless adapter for Kali Linux and joining the same SSID as the Pi. The wireless adapter must be placed into Monitor mode for traffic capture.

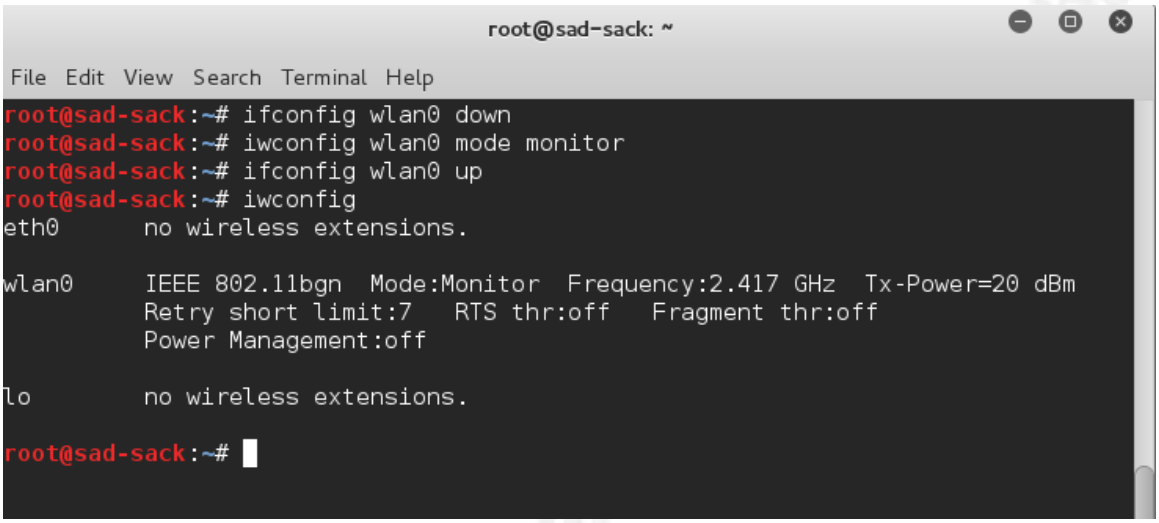

Figure 16 Putting the Kali wireless adapter into Monitor mode

By joining the wireless network, and utilizing Wireshark, we can capture packets and export using the Raspberry Pi's MAC ID as a filter. By default, the packets will only show as 802.11 encrypted packets.

Michael Horkan,
mhorkan4223@gmail.com

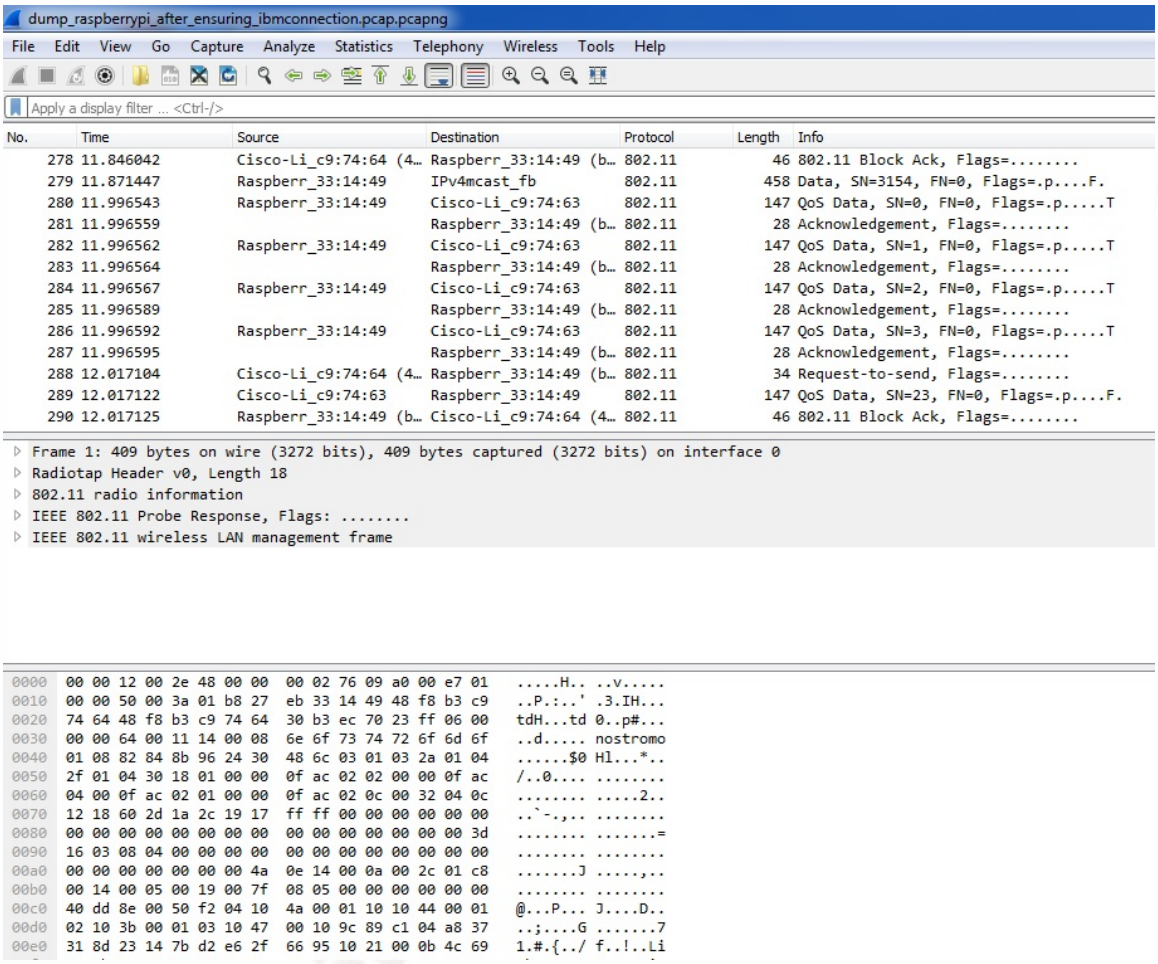

Figure 17 Raspberry Pi wireless traffic as 802.11 protocol

The packets displayed are decrypted by entering the wireless access point's pre-shared key into Wireshark's Edit > Preferences > Protocols > IEEE 802.11 > Edit Decryption Keys, selecting "Enable Decryption", and reloading (CTL-R). It may be necessary to toggle the "Assume packets have FCS" setting on the IEEE 802.11 (I had to do so to decrypt the traffic in my example).

Now one can see the application-layer traffic going between the Pi and wireless router. Filtering on "tcp" gives this result:

Michael Horkan,
mhorkan4223@gmail.com

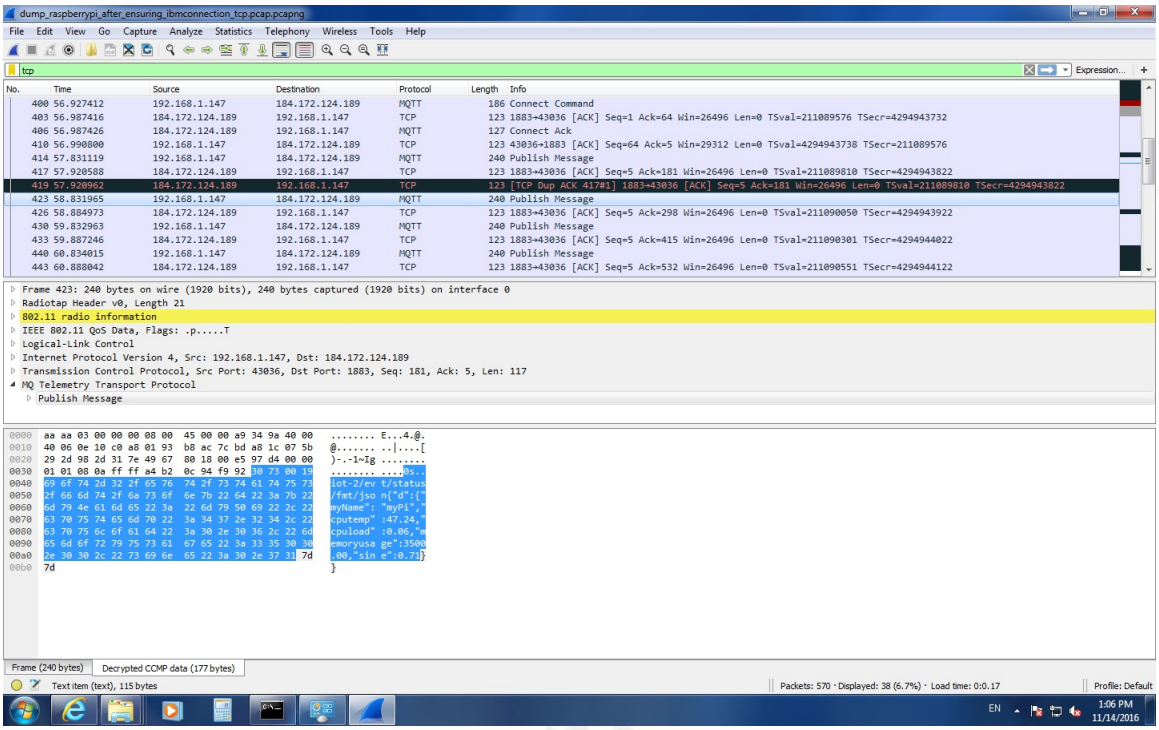

Figure 18 Decrypted Pi IoT TCP traffic

The IoT status data that the Pi is sending to the IBM cloud is visible here in "MQTT" application traffic. This is MQ Telemetry Transport, a light-weight publisher/subscriber messaging protocol designed for Internet of Things devices. (MQTT.org, n.d.)

Within the decrypted packets that the data sent to the cloud is a JSON dictionary:

{"myName":"myPi",

"cputemp":<float value>,

"cpuload":<float value>,

"memoryusage":<float value>,

"sine":<float value>}

From the cloud to the client device, there is a connection acknowledge message in MQTT. There are no other packets flowing in this direction (apart from TCP Acknowledges). The JSON dictionary above could be easily modeled in a fuzzing tool such as Sulley. This would be a network fuzzing test against the cloud server itself. Fuzz

Michael Horkan,
mhorkan4223@gmail.com

testing the client IoT device would require an application that sends a data payload to the client, and the modeling of those packets within the fuzzing tool.

MQTT does not appear to have any support for encryption. As it is meant for devices with relatively low hardware specifications, MQTT relies on standard, lower-layer encryption protocols such as TLS/SSL to supply that functionality (HiveMQ, 2016).

This particular IoT application does not employ any form of authentication, which is unsurprising considering that its purpose is as a turn-key learning tool. This makes this application vulnerable to impersonation attacks, whereby an attacker could set up their own script or application to pretend to be either the client or the cloud server.

Michael Horkan,
mhorkan4223@gmail.com

7. References

HiveMQ. (2016). *Introducing the MQTT Security Fundamentals.* Retrieved from HiveMQ Enterprise MQTT Broker: www.hivemq.com/blog/introducing-the-mqtt-security-fundamentals

IBM Developer Recipes. (2015, February 25). *Connect a Raspberry Pi To IBM Watson IoT Platform.* Retrieved from developerWorks Recipes: https://developer.ibm.com/recipes/tutorials/raspberry-pi-4/

MQTT.org. (n.d.). *MQTT.org Front Page*. Retrieved from mqtt.org: mqtt.org

Rippel, E. (2016). *Security Challenges in Embedded Designs.* Retrieved from Design & Reuse: http://www.design-reuse.com/articles/20671/security-embedded-design.html

Searle, J. (n.d.). *NESCOR Guide to Penetration Testing for Electric Utilities.* Retrieved from Electric Power Research Institute, Smart Grid Resource Center: http://smartgrid.epri.com/doc/NESCORGuidetoPenetrationTestingforElectricUtilities-v3-Final.pdf

Michael Horkan,
mhorkan4223@gmail.com