

Design and Implementation of a RADIUS Packet Generator ¹

Prabhaker Mateti and Ben Murray
Wright State University
Dayton, Ohio 45435
<http://www.cs.wright.edu/~pmateti/>

Working Draft/ October 10, 2006

¹This work was supported in part by a grant from Cisco Systems CIAG (Critical Infrastructure Assurance Group).

Contents

1	Introduction	1
1.1	My Python Style	1
1.2	Example Clients	2
1.2.1	The Simplest Client	2
1.2.2	A Client with Own User-Password	3
1.2.3	A <code>radtest</code> -like Client	4
1.2.4	<code>libpcap</code> dumpfile example	6
1.2.5	Pre- and Post-Actions	7
1.2.6	Constants	7
1.3	Comparison With Other Tools	8
1.3.1	Scapy	8
1.4	Overview of What Needs to be Done	8
2	Interesting RADIUS Packets	10
2.1	Classification of RADIUS Packets	12
2.2	Regenerating Packets	12
2.3	Non-“Access-Request”	14
2.4	Malformed	14
2.4.1	Arbitrary UDP packets	14
2.4.2	Invalid Attributes	15
2.4.3	Multiple Attributes	19
2.5	Unusual But Well-Formed Packets	20
2.6	Temporally Semi Valid RADIUS packet	21
2.7	Limitations of This Package	21

2.7.1	Globally Unique Request Authenticators	22
2.7.2	Temporal Uniqueness	22
2.7.3	Unpredictably Random Numbers	22
3	Class RadiusTest	24
3.1	Sequences of Packets	25
3.1.1	Test Actions	25
3.2	Random Packet Triplets	27
4	RADIUS Definitions	29
4.1	The RADIUS Definitions File	31
4.1.1	Conversion from dictionaries	32
4.2	Reading The Attribute Definitions	32
4.3	Using Definitions	35
4.4	Standard RADIUS Definitions	36
4.5	Scratch and Dent	36
5	Client	37
5.1	Channel	39
5.1.1	RADIUS packet Construction	44
5.1.2	RADIUS Communication	45
6	RADIUS Packets	49
6.1	RADIUS Packets	50
6.1.1	RADIUS Packet Initialization	51
6.1.2	Composition and Generation of Packets	56
6.1.3	Methods of RadiusPacket	57
6.2	RADIUS Identifier	62
6.3	Radius Authenticator	63
7	Packet Utilities	65
7.1	RADIUS Packet Dumps	66
7.2	General Packet Generators	72
7.3	Lower-Layer Packets	73

8 Attributes	78
8.1 Known Attributes	79
8.2 Storing Attributes	81
8.3 Choosing Attributes and Values	86
9 Attribute Generation	90
9.1 Basic Type Generators	90
9.2 General Value Generators	93
10 Logging and Monitoring	94
10.1 Logging	94
10.2 Monitoring	95
11 Utility Functions	96
12 Conclusion	103
13 References	104
References	104
A GUI Script Creator	105
B Testing Many RADIUS packets in Parallel	125
C Testing FreeRADIUS authenticator handling	128

Abstract

This report documents the design and implementation in Python of a RADIUS packet generator in `noweb`. The package is as yet incomplete. But we intend it to provide all the functionality of `pyrad` and `py-radius`.

Chapter 1

Introduction

This report documents the design and implementation in Python of a RADIUS packet generator in `noweb`. The package is as yet incomplete. But we intend it to provide at least the functionality of `pyrad` and `py-radius`.

Our focus here is not RADIUS accounting, but the “pure” protocol as defined in RFC 2865. By pure, we mean that we are not interested in vendor related attributes.

However, keep in mind that some one may extend this package in future to include accounting also.

More to be written.

1.1 My Python Style

In the code, I use `mixedCase` identifiers. I find ids with underscores ugly.

I want all my Python defs and classes to end with `pass` or `return`. I am insecure about code being accidentally deleted while editing. For the same reason, my files end with `-eof-`.

Wherever possible, methods/functions are (already or intended to be) documented in the prose with pre- and post-conditions. I do not care so much about doc strings, for the moment.

I do not use the `from module import *` version preferring instead `import module`. Seeing the module name prefix helps me.

TBD{ Wherever possible, make field names and methods “private” using the starts-with-underscore convention. }

TBD{ Single instances of classes, as in `Util.tightlyPack()` should be fixed. }

All exceptions should derive from `util.PktGenException`. All RADIUS exceptions should derive from `util.RadiusException`. Similarly, all warnings should derive from `util.PktGenWarning`. TBD{ The exception hierarchy may be a little too flat now? }

1.2 Example Clients

Here are two simple example clients¹ using `radiuspktgen` that is documented in this article.

1.2.1 The Simplest Client

```
<clientExample0.py>≡
#!/usr/bin/env python
# clientExample0.py generated from intro.nw

from radiuspktgen.channel import Channel
achannel = Channel("radiusServer.osis.wright.edu",
                  "sharedSecret")
achannel.conductTest()

# end of test
```

¹Note to myself: redo

1.2.2 A Client with Own User-Password

In the following test, the tester choses to override the generation of User-Names and User-Passwords.

```

<clientExample1.py>≡
#!/usr/bin/env python
# clientExample1.py generated from intro.nw
from radiuspktgen.channel import Channel
from radiuspktgen.util import random

class MyGenFunctions:
    def __init__(self):
        self.unmpwd = [("pmateti", "pmateti0123"),
                       ("bmurray", "ben987654"),
                       ("pmateti", "another-password-pmateti0123") ]
        self.randomx = 0
        # Map attribute names to those value-generating functions (closures!)
        self.gen = {"User-Name": self.generateMyKindOfName,
                   "User-Password": self.generateMyKindOfPassword,
                   "attributeNamesGen": ["User-Name",
                                         "User-Password"]}

    def generateMyKindOfName(self):
        self.randomx = random.randint(0, len(self.unmpwd)-1)
        return self.unmpwd[self.randomx][0]

    # MD5 encryption is still done internally
    def generateMyKindOfPassword(self):
        return self.unmpwd[self.randomx][1]
    pass

achannel = Channel("radiusServer.osis.wright.edu",
                  "sharedSecret")
achannel.defineActions(MyGenFunctions().gen)
achannel.conductTest()

# end of test

```

1.2.3 A radtest-like Client

This is an incomplete imitation of the “radtest” program commonly supplied with FreeRADIUS and other Livingston-descended servers. By this we mean that it accepts the same command-line arguments (even if it does not use all of them), and sends one or more RADIUS packets generated from this information. We will explain this client in detail, because it exemplifies the basic format for test scripts written with our generator.

In our packet generator, `Channel` (chapter 5) objects represent a network connection between the RADIUS client and a single RADIUS server. Numerous “actions” can be given values on a channel to affect request generation (see `channel.defineActions`, §5.1.0.1). Tests are run by calling a channel’s `conductTest` method. This takes a list of (packet, count, delay) triplets as discussed in chapter 3. The definition of a test, then, consists of setting up a channel and these three values.

In general, a “value” can be either constant or a function. This is because many values will be used many times: this `packet` value will be used `count` times, and the request attributes will be used once per request. A constant value will always be used unchanged (the attribute values given in the packet constructor’s `attrNameValueList` will be the same for each request), but a function can close over some state variables and thus return new values when desired.

This test will send a request over a channel, so it creates `ChannelPackets` (see chapter 6 for more differences between classes representing RADIUS packet). Request attribute values can be added either in the `attrNameValueList` (as here) or by channel actions. For simplicity, the command line values are used with only minimal checking.

```
<clientExampleRadtest.py>≡
#!/usr/bin/env python
# clientExampleRadtest.py generated from intro.nw
import sys

if len(sys.argv) < 6:
    print("Usage: %s user passwd radius-server[:port] "
          "nas-port-number secret [ppphint] [nasname]" % sys.argv[0])
    sys.exit()

from radiuspktgen import util
from radiuspktgen.channel import Channel
from radiuspktgen.packet import ChannelPacket
achannel = Channel(server=sys.argv[3],
                   sharedSecret=sys.argv[5])

try:
    port = sys.argv[3].index(":")
    achannel.authport = sys.argv[3][port+1:]
```

```
s = sys.argv[3][:port]
except ValueError:
    port = 1812

if len(sys.argv) == 8:
    nasname = sys.argv[7]
else:
    nasname = util.thisHostIpAddress()

if not util.isHostReachable(achannel.server):
    print("The server %s cannot be reached." % achannel.server)
    sys.exit(1)
    pass

def genPacket():
    return ChannelPacket("aqm", achannel,
        [("User-Name", sys.argv[1]),
         ("User-Password", sys.argv[2]),
         ("NAS-Port", sys.argv[4]),
         ("NAS-IP-Address", nasname)
        ])

achannel.conductTest([(genPacket, 1, 0)])

# end of test
```

1.2.4 libpcap dumpfile example

This script uses the libpcap dumpfile reader and writer (see §7.1) to read all RADIUS messages with an identifier of 160 from the capture file given by the first command line argument. It then writes a new capture file (with filename given by the second argument) containing these messages stripped of their passwords.

Note that messages are dissected literally, so the automatic attribute length generation is turned off. This script turns it back on for the password attribute so it's length is updated after a new value is given. An alternative would have been to manually update the literal value.

```
<examplePcap.py>≡
#!/usr/bin/env python
# examplePcap.py generated from intro.nw
import sys

from radiuspktgen.attribute import StandardRadiusDefs
from radiuspktgen.packetUtil import RadiusDumpReader, RadiusDumpWriter

if len(sys.argv) < 3:
    print("Usage: %s in.cap out.cap" % sys.argv[0])
    sys.exit()
    pass

f = RadiusDumpReader(sys.argv[1])
o = RadiusDumpWriter(sys.argv[2])
for p in f.byId(160):
    for attr in p.attributes:
        if attr.code == StandardRadiusDefs["User-Password"].code:
            attr.autoLength = True
            attr.value = ""
            pass
        pass
    o.writePacket(p)
    pass
pass
```

1.2.5 Pre- and Post-Actions

The following prints the packet sent and the response received.

<be Verbose>≡

```
packetID = None

def preSend():
    packetID = achannel.last.identifier
    ptext = str(achannel.last)
    print("packet to be sent:", self.packet, time.ctime(), "\n")

def postDelayAction():
    replyPacket = achannel.recv()
    print("reply received:", replyPacket, time.ctime(), "\n")
```

1.2.6 Constants

Several constants referenced throughout our library are defined here.

<intro.py>≡

```
# intro.py generated from intro.nw

RAND_ATTR_NAME = "<Random>"

StandardRadiusDefsFilename = "radius-defs.txt"
def setRadiusDefsFilename(filename):
    global StandardRadiusDefsFilename
    StandardRadiusDefsFilename = filename

# -eof-
```

1.3 Comparison With Other Tools

Unlike most packet generators, the tool described here is designed to function as a full-fledged application “out of the box” for the RADIUS protocol (and any protocols we may add later). A test written with another general-purpose tool might be built out of the basic operations and protocol data structures provided by the tool—as if the author were writing a new tool every time. By contrast, this program provides the complete application so tests can be written by hooking into or replacing the provided application only where needed.

Generated packets may be valid or invalid. We have tried to provide hooks where ever it is convenient to munge values, and we also include protocol-specific logic with which authors can ensure validity.

We provide a GUI test creator to introduce users to the tool [A](#). Useful tests can be written entirely with the GUI ([C](#) could be re-created, for example). Since its output is simply a Python script like any other for our tool, the GUI can also be used as a starting point for complex manipulation of packets.

1.3.1 Scapy

Scapy [[Biondi 200x](#)] is another packet manipulator written in Python. It is intended as a framework for general-purpose network analysis. As Scapy claims to replace “hping, 85% of nmap, ... etc” in addition to creating or dissecting packets from various protocols, functionality for tasks beyond the scope of our tool such as scanning, tracerouting, and graphing is included.

Numerous protocols have long been supported by Scapy, and thus has a more mature system for defining data structures. Scapy’s facilities for generating generic data like IP addresses are likewise advanced, but it does not include protocol-specific logic (e.g. which combinations of optional fields an RFC specifies are valid).

1.4 Overview of What Needs to be Done

This section contains a list of to-do items. It *is* a very short term (hours and days) list.

Make sure there are no shallow/deep copy problems.

Explore Python `unit test`.

Maintain the packet collections mentioned in [§10](#).

Provide a way to define named values (non-consecutive integers e.g. “Acct-Status-Type”).

Make this whole thing a **package**. Having a module is a start:

```
<__init__.py>≡  
# __init__.py generated from intro.nw  
__all__ = ["channel",  
          "attribute",  
          "packet",  
          "defs",  
          "packetLow",  
          ]  
  
# -eof-
```

[Interlink Networks]

Chapter 2

Interesting RADIUS Packets

This chapter drives the design of our PGL (packet generation language).

Past experience proves that typical RADIUS servers are not robust even with respect to malformed packets. In our expectation, the interesting packets are temporally semi-valid packets that are “confusing” to the RADIUS server in some sense.

Below we cook up some examples of interesting packets using the library described in these chapters.

TBD{ Code yet to be written. This chapter yet to be written properly. Right now, it is a bunch of notes. }

```
<interesting.py>≡
#!/usr/bin/env python
# interesting.py generated from interesting.nw
<main>
from copy import copy
import struct
import random

from radiuspktgen import attributeGen
import radiuspktgen.util

<Change UDP Packet>
<Illegal Packet Code>
<Minimal Packet>
<Authenticator Absent or Incomplete>
<Generate Empty User-Name>
<Unusual User-Password Lengths>
<Disallowed Attribute Lengths>
<ByteSeq Too Short>
```

<Extraneous Bytes>
 <One Byte Remaining>
 <Duplicate User-Name>
 <Generate No User-Password>
 <Overlong RP>

We illustrate how to set various generators for each interesting case. See also the examples of `intro.nw`.

```

<main>≡
import sys

if (len(sys.argv) < 6):
    print("Usage: %s user passwd radius-server nas-port-number "
          "secret [ppphint] [nasname]" % sys.argv[0])
    sys.exit()

from radiuspktgen.channel import Channel
from radiuspktgen.packet import ChannelPacket
import radiuspktgen.attribute

user = {"User-Name": sys.argv[1],
        "User-Password": sys.argv[2]}

def interestingTest(actionsDict):
    achannel = Channel(sys.argv[3],
                      sys.argv[5])
    achannel.defineActions(user)
    def genPacket():
        return ChannelPacket("aqm", achannel)

    achannel.defineActions(actionsDict)
    print("\nConducting Interesting Packet Test:")
    for name, action in actionsDict.items():
        print("%(name)s: %(action)s" % locals())
    print
    achannel.conductTest([(genPacket, 1, 0)])
  
```

2.1 Classification of RADIUS Packets

[Mateti 2005] rigorously defines the following subsets of IP packets using OM. Here they are rendered in Python.

Non-RADIUS packet These are irrelevant to us in this report.

Well-formed Packets These are valid in a context-free sense. Within the one packet in question, individual fields and interrelationships among fields are all what they should be.

Malformed Packets are syntactically invalid. E.g., a packet that contains two User-Name fields, or a packet whose length is incorrect are malformed.

Temporally Fully Valid packets obey all the requirements of the ERFC 2865, which, of course, include being well-formed and constraints based on history, such as temporal uniqueness of authenticator values.

Temporally Semi Valid packets are fully valid packets *if* we ignore portions of the history of either RADIUS client or RADIUS server or both.

A normal (i.e., non-malicious) RADIUS client would be sending Access-Request messages that are semi valid.

2.2 Regenerating Packets

Read from a file of dumped packets, and make a generator of RADIUS packets (may be IP packets carrying the UDP carrying the RADIUS packet). The dump is typically a capture of actual packets between a RADIUS client and a RADIUS server. The dump can also be hand-crafted. The packets are typically well-formed, but need not be.

Must be able to alter IP addresses, which is beyond the scope of our current generator/test harness, because e.g. changing the destination requires a new `Channel`. Not to mention sending packets with an arbitrary source IP/port.

`changeUdpPacket` changes the non-None fields of the `udpPacket`.

<Change UDP Packet>≡

```
def changeUdpPacket(sourceIpAddress=None, sourcePort=None,
                    destIpAddress=None, destPort=None):
    def changer(udpPacket):
        if sourceIpAddress is not None:
            udpPacket.network.sourceIP = sourceIpAddress
            pass
        if sourcePort is not None:
            udpPacket.sourcePort = sourcePort
```

```
        pass
    if destIpAddress is not None:
        udpPacket.network.destIP = destIpAddress
        pass
    if destPort is not None:
        udpPacket.destPort = destPort
        pass
    return pktbytes
return changer

if os.path.exists("radius.cap"):
    from radiuspktgen.packetUtil import RadiusDumpReader, RadiusDumpWriter
    cap = RadiusDumpReader("radius.cap")
    changer = changeUdpPacket(sourceIpAddress="127.0.0.1")
    changedPackets = [ changer(p) for p in cap.findIf(True) ]
    out = RadiusDumpReader("radius-changed.cap")
    for p in changedPackets:
        o.writePacket(p)
        pass
    pass
```

2.3 Non-“Access-Request”

A RADIUS server should be tested by sending it non- Access-Request but otherwise well-formed RADIUS packets.

2.4 Malformed

Several malformed packet constructions are shown, each with a different goal. All these can also be combined. TBD{ Check }

2.4.1 Arbitrary UDP packets

1. A RADIUS server should be tested by sending it arbitrary UDP packets. These are the most obvious among the malformed packets.
2. Illegal code values in the RADIUS packet.

```

<Illegal Packet Code>≡
def illegalPacketCode(pktbytes):
    # Python strings are immutable, so create a new string
    # which is a random code byte followed by the remainder
    # of the old bytes.
    pktbytes = (chr(attributeGen.randomByte(14, 254))
                + pktbytes[1:])
    return pktbytes

interestingTest({"pktbytesManipulation": illegalPacketCode})

```

FreeRADIUS behaves as follows: Error: WARNING: Bad RADIUS packet from host 192.168.17.210: unknown packet code <n>, where <n> is the code number.

3. Send a short RADIUS packet with exactly 20 bytes, valid message code, identifier, length, and authenticator fields, but no attributes at all.

FreeRADIUS behaves as follows: Error: rlm_sql (sql): zero length username not permitted

```

<Minimal Packet>≡
interestingTest({})

```

4. Send a way-too-short RADIUS packet, with 4 .. 19 bytes, with valid message code, id, and length fields, but authenticator absent or incomplete.

FreeRADIUS behaves as follows: the short length (e.g. 16 or 4) is reported to be less than the minimum 20 bytes: `Error: WARNING: Malformed RADIUS packet from host 192.168.17.210: too short (received 16 < minimum 20)`

```

<Authenticator Absent or Incomplete>≡
def mPktTruncAuthenticator(pktbytes):
    assert len(pktbytes) == 20 # start with a minimal packet
    return pktbytes[0:-4]
interestingTest({"pktbytesManipulation": mPktTruncAuthenticator})

def mPktDropAuthenticator(pktbytes):
    assert len(pktbytes) == 20 # start with a minimal packet
    return pktbytes[0:-16]
interestingTest({"pktbytesManipulation": mPktDropAuthenticator})

```

5. Send a too-short RADIUS packet, which has a valid message code, and id, but length field value is greater than the actual length of the packet.
 - (a) With length, authenticator, and a few attributes.
 - (b) Same as (a), but dropping the last attribute value bytes (and without changing the length afterward).

2.4.2 Invalid Attributes

1. The following sets User-Name generator so that the values are empty.

```

<Generate Empty User-Name>≡
def generateEmptyUserName():
    return ""

interestingTest({"User-Name": generateEmptyUserName,
                "attributeNamesGen": ["User-Name"]})

```

FreeRADIUS behaves as follows: Caught at the module level; do all modules check? Error: rlm_sql (sql): zero length username not permitted

2. Generate User-Passwords with length outside the [16 .. 128] range.

```

<Unusual User-Password Lengths>≡
def generateLongUserPassword():
    return attributeGen.randomByteSeq(129, 150)

interestingTest({"User-Password": generateLongUserPassword,
                "attributeNamesGen": ["User-Password",
                                     "User-Name"]})

def generateShortUserPassword():
    return attributeGen.randomByteSeq(1, 15)

interestingTest({"User-Password": generateShortUserPassword,
                "attributeNamesGen": ["User-Password",
                                     "User-Name"]})

```

FreeRADIUS behaves as follows: FreeRADIUS accepted a 140-character string.

3. Construct attributes that must be of fixed length (such as 4-byte unsigned integers) so that their length is not equal to that fixed length.

FreeRADIUS behaves as follows: No warning message is printed, but the attribute is decoded as a hex number in the logs instead of an integer. The other attributes are processed normally, possibly resulting in an AAM.

```

<Disallowed Attribute Lengths>≡
def tooLongNASPortValue():
    return attributeGen.randomByteSeq(16)
interestingTest({"NAS-Port": tooLongNASPortValue,
                "attributeNamesGen": ["User-Password",
                                     "User-Name",
                                     "NAS-Port"]})

def tooShortNASPortValue():
    return attributeGen.randomByteSeq(2)
interestingTest({"NAS-Port": tooShortNASPortValue,
                "attributeNamesGen": ["User-Password",
                                     "User-Name",
                                     "NAS-Port"]})

```

4. After processing the last attribute, make the left-over byte sequence too short compared to the total length of the RADIUS packet.

FreeRADIUS behaves as follows: Discarded, with log message for the lengths, e.g. Error: WARNING: Malformed RADIUS packet from host 192.168.17.210: received 26 octets, packet length says 45

```

<ByteSeq Too Short>≡
def byteSeqTooShort(pktbytes):
    x = radiuspktgen.util.randint(21, len(pktbytes)-1)
    pktbytes = pktbytes[:x]
    return pktbytes

interestingTest({"pktbytesManipulation": byteSeqTooShort,
                "attributeNamesGen": ["User-Password",
                                     "User-Name"]})

```

5. Extraneous Bytes: “Octets outside the range of the Length field MUST be treated as padding and ignored on reception.” [RFC 2865]

1

FreeRADIUS behaves as follows: extraneous bytes are silently ignored and the packet is processed normally.

```

<Extraneous Bytes>≡
def extraneousBytes(pktbytes):
    maxUDPlength = 65527
    addLength = (maxUDPlength - len(pktbytes))
    pktbytes += attributeGen.randomByteSeq(1, addLength)
    return pktbytes

interestingTest({"pktbytesManipulation": extraneousBytes,
                "attributeNamesGen": ["User-Password",
                                     "User-Name"]})

```

¹Ben: For maxUDPlength I figure 16 bits length field = 65535 bytes - 8 byte header?

Note that the UDP packet length is correctly set by `UDP send`.

6. After the last attribute, we are left with just one byte, according to the RADIUS packet length. See `dissect`. May be post `udpPayload`. After appending one extra byte, as in `extraneousBytes`, we need to increment the RADIUS packet length.

FreeRADIUS behaves as follows: `Error: WARNING: Malformed RADIUS packet from host 192.168.17.210: attribute 123 too short`

<One Byte Remaining>≡

```
def oneByteRemaining(pktbytes):
    oneByte = attributeGen.randomByte()
    length = (struct.unpack("!H", pktbytes[2:4]))[0] + 1
    pktbytes = (pktbytes[0:2]
                + struct.pack("!H", length)
                + pktbytes[4:]
                + chr(oneByte))
    return pktbytes
```

```
interestingTest({"pktbytesManipulation": oneByteRemaining})
```

7. After processing the last attribute, make the left-over byte seq just two bytes. That is, attribute value is empty, but attribute type is a valid number, and length == 2.

2.4.3 Multiple Attributes

1. Some attributes must not occur more than once. E.g., in an Access-Request message, multiple User-Names are not permissible.

FreeRADIUS behaves as follows: The first User-Name is honored, others are silently ignored.

```

<Duplicate User-Name>≡
duplicateUserNames = ["User-Name",
                      "User-Password", "Login-TCP-Port",
                      "User-Name"]

def twoUserNamesI(userName, last):
    userName2 = list(userName)
    userName2.reverse()
    userName2 = "".join(userName2)
    vals = [userName, userName2]
    if last:
        vals.reverse()
    for val in vals:
        yield val
<Duplicate User-Name: Same>
<Duplicate User-Name: Valid First>
<Duplicate User-Name: Valid Last>

<Duplicate User-Name: Same>≡
interestingTest({"attributeNamesGen": duplicateUserNames})

<Duplicate User-Name: Valid First>≡
twoUserNames = twoUserNamesI(sys.argv[1], False)
interestingTest({"attributeNamesGen": duplicateUserNames,
                "User-Name": (lambda : twoUserNames.next())})

<Duplicate User-Name: Valid Last>≡
twoUserNames = twoUserNamesI(sys.argv[1], True)
interestingTest({"attributeNamesGen": duplicateUserNames,
                "User-Name": (lambda : twoUserNames.next())})

```

2. The presence of an attribute *A.1* implies the presence of some other *A.2*.

FreeRADIUS behaves as follows: Silently rejected.

```

⟨Generate No User-Password⟩≡
  generateNoUserPassword = ["User-Name", "Login-TCP-Port"]
  interestingTest({"attributeNamesGen": generateNoUserPassword})

```

2.5 Unusual But Well-Formed Packets

1. A RADIUS packet must not exceed 4096 bytes, implying no more than 4076 bytes for attributes. Strain the RADIUS server by sending it packets that contain 2000+ attributes.

FreeRADIUS behaves as follows: **Error: WARNING: Malformed RADIUS packet from host 192.168.17.210: too long (length 12020 > maximum 4096)**

```

⟨Overlong RP⟩≡
  StandardRadiusDefs = radiuspktgen.attribute.StandardRadiusDefs
  manyAttrs = copy(StandardRadiusDefs.byname.keys())
  manyAttrs += ["Login-TCP-Port"] * 2000
  interestingTest({"attributeNamesGen": manyAttrs})

```

2. User-Name: arbitrary byte seq (outside of A-Za-z0-9), especially consisting of ascii control characters, blanks, tabs, vertical tabs, newline, cr, and bytes with most significant bit set to 1.
FreeRADIUS behaves as follows: "\v\t\ny\$\mu\$\cent" is accepted.
3. User-Password: arbitrary byte seq (as above). Past vulnerabilities include passwords of certain length or containing blanks.
4. Proxies are expected to keep the order of attributes even while consuming certain attributes.
5. The State attribute is expected to keep the state received and pass it back to a RADIUS server. Corrupt it.

2.6 Temporally Semi Valid RADIUS packet

This package can assist in testing temporal uniqueness.

A server may fail in a temporal sense and hence not reject a temporally semi valid RADIUS packet. In an on-going “session”, the identifier and/or authenticator is repeated. (The RFC uses the word “session” but only with respect to a user. Here I mean it with respect to a pair of RADIUS client and RADIUS server.)

1. Consecutive RADIUS packets are happening with fairly short (say, less than a minute) delay between them. Meanwhile, there is a flood of RADIUS packets from other RADIUS clients. The overall memory (plus disk space) required can be beyond available resources.
2. The time delay is very long (hours? days? months?). How old a memory should the RADIUS server have?
3. Suppose the RADIUS server rebooted. Must the record of what was seen in prior sessions be persistent across reboots?

2.7 Limitations of This Package

RFC 2865 requires that certain numbers in RADIUS, should be (i) *universally unique*, (ii) *temporally unique*, and/or (iii) unpredictably random. While these properties are important, this package has no built-in features to assist in testing (i) and (iii). Such testing has to be user supplied.

The terms (i) and (ii) are generally described loosely, and often imply that these two terms are equivalent. But, they are not. The essence of the difference is that (ii) refers to uniqueness in the context of what has transpired so far, whereas (i) is time independent, i.e., unique from “big bang” of the distant past to “apocalypse” of the unknown future.

It turns out that it is not too difficult to rigorously define the terms [?] using discrete mathematics and logic. But, it is possible that the RFC did not “really mean it.”

The difficulty with respect to (i) and (ii) is in arriving at a practical implementation that hopes to do justice to the two obligations: (a) the “sender” should generate such a number, and (b) the “receiver” should be algorithmically able to verify that the received number is one such. Depending on whether we chose interpretation (i) or (ii), the difficulties are different.

2.7.1 Globally Unique Request Authenticators

Traditionally, the problems of (i) are “solved” by using number granting authorities and associated query/answer protocols. Because these solutions are inadequate various spoofing attacks became possible.

Globally unique implies communication among *all* RADIUS clients even if different RADIUS servers are involved. Even if we restrict the universe to one RADIUS server (together with any proxies it may use) and all its RADIUS clients we do not see a feasible implementation.

2.7.2 Temporal Uniqueness

Note that this kind of uniqueness is obviously in the context of a group of “connected”² senders and receivers. The problems of (ii) are solvable (if we put aside the resource requirements) as follows. Every sender and receiver has access to a global historical record of what has transpired. Using such records, (a) and (b) can be implemented. But, the resource requirements – in particular, space to store the history, and the distributed computing mechanisms to keep it up-to-date across all hosts – are immense even when the group of connected hosts is small.

Temporal uniqueness has feasible, if space intensive, implementations as it relates to a pair of RADIUS client and RADIUS server.

2.7.3 Unpredictably Random Numbers

“Unpredictable” is arguably a mathematically undefinable concept. Note that to be able to speak of values in a probabilistic manner makes it predictable.

We presume that the intended meaning is that *for a third party, having observed a sequence of certain values of a field being exchanged between two parties, it*

²Not TCP-connected, but in the sense of hosts A and B are “connected” if they ever exchanged a message of the protocol, or transitively there is a third party C with which they did.

is computationally infeasible to compute what the future values of exchange will be.

“The phrase ”computationally infeasible” is used frequently in cryptography but is rarely defined. The general consensus on its meaning is as follows. If the time complexity of an algorithm A is a function that grows faster than any polynomial, we consider A to be computationally infeasible. A similar meaning with respect to memory (and other) resources required is included in the meaning of the phrase. On a practical level, we should understand the phrase to stand for any computation that requires either extremely long time or extremely high resource requirements even on the fastest (parallel, cluster, etc.) computer systems. Extremely long here is in the class of several (zillion?) years.” [?].

Note that the “unpredictable” property regarding a certain number in a field usually has further dependencies on other field values that are known, e.g., in the RADIUS protocol, the shared secret, the server id, and the client id that share this secret.

Chapter 3

Class RadiusTest

```
<radiusTest.py>≡
# radiusTest.py generated from radiusTest.nw
import time

from attribute import Attribute, StandardRadiusDefs
from packet import ChannelPacket
import channel
import util
from util import random as random

class RadiusTest:

    def __init__(self, channel):
        self.channel = channel
        self.packet = None
        # Custom actions from the channel, if any
        for action in channel.testActions:
            func = getattr(channel, action)
            if func is not None:
                setattr(self, action, func)

<randomPacketTriplets>
<Default Actions>
<conductTest>

<main>
# -eof-
```

3.1 Sequences of Packets

An example: All Access-Request packets. One hundred valid ones + one empty user name + random number but less than 50 valid ones + two arbitrary UDP packets + twenty valid ones.

The need to express such sequences drives the design of syntax of our PGL (packet generation language). For now, we need a compact way of expressing these in Python.

A seq of packets is implemented as a finite/infinite generator. Suppose `g0`, `g1`, ... are single packet generators. The bottom line spec for these generators is that they must generate a UDP payload, that may or may not be a well-formed RADIUS packet.

`q = [(gi, ni, di) | ..]` specifies the generation of $ni \geq 1$ packets by `gi`. The PGL for it may look like `gi ^ ni @ di + gj ^ nj @ dj` and `^ ni` can be omitted when `ni == 1`, `@ dj` can be omitted when `dj == 0`.

These `gi` function currently takes no parameters, but by necessity it will need to use “know” about the channel on which it is to be sent (use the same shared-Secret, etc.). See `clientExampleRadtest` (§1.2.3) for an example.

3.1.1 Test Actions

Test actions are instance variables which should be set to methods (functions with the first parameter being `self`). A default is supplied for each. The user may supply replacements as desired.

TBD{ Document the below if/where their purpose is not clear. }

The following test actions are invoked during `conductTest`. `util.invoke` is used to robustly call each method.

- `preTest`
- `postTest`
- `postDelayAction`
- `chooseAttr` (must return an `AttributeDef` object)
- `genCount`
- `genDelay`

After the generation of a packet, the default action is to send the packet to pre-selected RADIUS server and wait for a corresponding reply. A tester is able to customize this action.

TBD{ Merge this into intro? e.g., near beVerbose? }

⟨Example customization⟩≡

```
def newPreTest(self):
    return someCondition(self.packet)

t = RadiusTest(achannel)
t.preTest = newPreTest
t.conductTest()
```

Note that the supplied method has full access to the following state of the RadiusTest object. See the default postDelayAction for an example of how this can be used.

channel: See §5.1.

attributes: See §8.1.

packet: The current packet.

⟨Default Actions⟩≡

```
preTest = None

postTest = None

def defaultPostDelayAction(_, self):
    replyPacket = self.channel.recv()
    if replyPacket is not None:
        source = "%s:%i" % (replyPacket.sourceIP, replyPacket.sourcePort)
        print("Reply received %s from %s:" % (time.ctime(), source))
        print(replyPacket)
        pass
    pass
postDelayAction = defaultPostDelayAction

def defaultChooseAttr(self):
    name = (self.channel.attributes.generateRandomAttributeName(1))[0]
    return StandardRadiusDefs[name]
chooseAttr = defaultChooseAttr

defaultGenCount = 1
genCount = defaultGenCount

def defaultGenDelay(self):
    return random.randint(1, 100)
genDelay = defaultGenDelay
```

3.2 Random Packet Triplets

`randomPacketTriplets` is a generator of (genPacketFunc, count, delay) triplets. It invokes the `chooseAttr`, `genCount` and `genDelay` test actions.

The following instance variables are methods invoked during `conductTest`. `util.invoke` is used to robustly call each method.

TBD{ Ideally each packet would have more than one attribute! }

$\langle randomPacketTriplets \rangle \equiv$

```
def randomPacketTriplets(self):
    gen = self.channel.attributes.generate
    while True:
        attr = util.invoke(self.chooseAttr)
        func = lambda : \
            ChannelPacket("aqm", self.channel,
                          [(attr.name,
                            gen(attr.code))])
        yield (func,
              util.invoke(self.genCount),
              util.invoke(self.genDelay))
    pass
```

The `lst` is a sequence of `(gi, ni, di)` triplets discussed above. It will almost always be given. When it is not, we generate a sequence of random triplets.

`<conductTest>`≡

```
def conductTest(self, lst=None):
    if lst is None:
        lst = self.randomPacketTriplets()
    invoke = util.invoke

    invoke(self.preTest)

    for (func, count, delay) in lst:
        for i in xrange(int(count)):
            self.packet = util.invoke(func)

            self.channel.send(self.packet)

            d = int(delay)
            if d < 0:
                d = self.genDelay
            time.sleep(d)
            invoke(self.postDelayAction, self)

    invoke(self.postTest)
```

`<main>`≡

```
if __name__ == "__main__":
    from channel import Channel
    achannel = Channel("radiusServer.osis.wright.edu", "sharedSecret")
    RadiusTest(achannel)
    pass
```

Chapter 4

RADIUS Definitions

RADIUS packet structure and several of its properties are captured in a file named `radius-defs.txt`. A description and examples are given in Section 4.1.

```
<defs.py>≡
# defs.py generated from defs.nw
import intro
import attributeGen
import util
from util import random as random

<class AttributeDef>

class RadiusDefs:
    unread = True

    def __init__(self, filename=None):
        self.byname = {}
        self.bynum = {}
        self.gen = [util.noop]*256
        self.isa = isa=[util.noop]*256

        if filename is not None:
            self.readDefsFile(filename)
        pass

<readDefsFile>
<consGenerateFunc>
<consIsValidFunc>

<Methods of RadiusDefs>
```

```
pass
⟨StandardRadiusDefs⟩
```

```
⟨main⟩
# -eof-
```

4.1 The RADIUS Definitions File

The format of the `radius-defs.txt` file is illustrated below. Each line describes one attribute, and it has the following fields in the order shown.

0	Name	Code	Type	Size	min	max	AQM	AAM	AJM	ACM
attribute	User-Name	1	string	1	255	<=1	<=1	==0	==0	==0
attribute	User-Password	2	string	1	255	<=1	==0	==0	==0	==0
attribute	Service-Type	6	uint	16	16	<=1	<=1	==0	==0	==0
attribute	Idle-Timeout	28	uint	16	16	==0	<=1	==0	==0	<=1

Name: The attribute name exactly as spelled out in the RFC.

Code: The attribute code number as assigned by the RFC.

Type: string, byte, or uint.

Size: min max; Size is the space in byte units occupied by the value. Always, min <= max. If it is fixed in length, min == max.

Relational Strings: These describe constraints to be satisfied among all attributes of a RADIUS packet.

Values We expect to add this entry. (i) Enumerate the possible values. (ii) Give a predicate that yields true on valid values from its type. The predicate is typically unary on a value from the type, but can name other parameters. (iii) Compute it from other values. The other values can be those of the parent (and grandfather, ...) object. Note that (i) is a special, but frequent, case of (ii). So is (iii).

Relational strings such as “<=1” are interpreted by `RadiusPacket.isValid()`. E.g., “<=1” stands for “this attribute must occur at most once”. Note that these are not arbitrary relations. The only possible relational strings are: >=0 ==0 ==1

<=1. ¹ For example: if `aqm == '>=0'` for attribute number `an` it implies that attribute numbered `an` may appear any number of times in an Access-request-Message. We use the following abbreviations:

AQM: Access-request-Message.

AAM: Access-Accept Message.

AJM: Access-reject Message.

ACM: Access-Challenge Message.

TQM: Accounting-Request Message.*

¹We could use shorter strings, but the mnemonic value is worth it.

TRM: Accounting-Response Message.*

TSM: Accounting-Status Message.*

2

4.1.1 Conversion from dictionaries

The `dict2defs.pl` script converts the “dictionary” file of FreeRADIUS, pyrad, etc. into this format. Because the FreeRADIUS dictionary format is missing some info (size, etc.), when adding previously unknown attributes `dict2defs.pl` will print a warning and set all constraints “==0”. The remaining information must then be adjusted.

Possibly cache generated attribute functions (from `consGenerateFunc` and `consIsValidFunc`) to the file named `generated-defs.py`.

4.2 Reading The Attribute Definitions

^{2*} Not yet supported.

The method `readDefsFile` reads a `radius-defs.txt` file, and constructs a table of tokens, and generated names of two functions. Currently, it is limited to reading attribute defs only.

This will raise an `IOError` if filename cannot be opened for reading.

```

<readDefsFile>≡
def readDefsFile(self, filename):
    ifd = open(filename, "rt")

    for line in ifd.xreadlines():
        line = line.split("#", 1)[0].strip()
        tokens = line.split()
        if not tokens:
            continue

        if tokens[0] == "attribute":
            adef = AttributeDef(*tokens[1:])
            self.byname[adef.name] = adef
            self.bynum[adef.code] = adef
            self.gen[adef.code] = self.consGenerateFunc(adef)
            self.isa[adef.code] = self.consIsValidFunc(adef)

    ifd.close()
    self.unread = False
    pass

```

Given the maximum and minimum allowable sizes, `consGenerateFunc` constructs a function to generate values. Various value generation components from §9 (such as `randomByteSeq` are used.

Currently, `consGenerateFunc` can always generate a byte sequence and `Attribute.set_value` will do the right thing for the attribute's type. TBD{ This approach hasn't been tested completely, so we may not be able to keep it so simple. In particular, named values would need handling if we could do them. }

```

<consGenerateFunc>≡
def consGenerateFunc(self, adef):
    func = (lambda : attributeGen.randomByteSeq(adef.szmin, adef.szmax))

    return func

```

`consIsValidFunc` works analogously to `consGenerateFunc`. “Valid” currently means only that the length is allowed.

```
<consIsValidFunc>≡
def consIsValidFunc(self, adef):
    func = (lambda length, value:
            (length > adef.szmin) and (length < adef.szmax))
    return func
```

Each object of the `AttributeDef` class represents a single attribute definition of. This class exists to avoid passing the data around in specially ordered tuples.

Each field corresponds to a column in the definition file. See §radiusDefs for their meanings.

This should be the only place where the column order is interpreted. Everywhere else can get this information from an object of this class without needing to “know” the order.

```
<class AttributeDef>≡
class AttributeDef:
    def __init__(self, name, code, type,
                szmin, szmax,
                aqm, aam, ajm, acm):
        self.name = name
        self.code = int(code) % 256
        self.type = type
        self.szmin = int(szmin)
        self.szmax = int(szmax)
        self.aqm = aqm
        self.aam = aam
        self.ajm = ajm
        self.acm = acm
        if "==" == aqm == aam == ajm == acm:
            print("Warning:", name, "is disallowed from all packet types")
            pass
        pass
    pass
```

4.3 Using Definitions

⟨Methods of RadiusDefs⟩≡
⟨getitem⟩
⟨has_key⟩
⟨iter⟩
⟨numOf⟩
⟨nameOf⟩

`__getitem__` maps both an attribute's name and number to its `AttributeDef` object. If the key satisfies `isdigit()` it is assumed to be an attribute code; otherwise the key is an attribute name, spelled exactly as in the definitions file. (Depending on the key, either `self.byname` or `self.bynum` dictionary is used.)

Note the assumption that no attribute name will consist entirely of digits.

⟨getitem⟩≡

```
def __getitem__(self, key):
    if isinstance(key, int) or key.isdigit():
        return self.bynum[int(key)]
    else:
        return self.byname[key]
```

`has_key` follows the same logic as `__getitem__`.

⟨has_key⟩≡

```
def has_key(self, key):
    return self.bynum.has_key(key) or self.byname.has_key(key)
```

⟨iter⟩≡

```
def __iter__(self):
    for adef in self.byname.values():
        yield adef
```

`numOf` and `nameOf` are convenience functions to convert between attribute code numbers and names.

⟨numOf⟩≡

```
def numOf(self, attributeName):
    if isinstance(attributeName, int):
        return attributeName
    else:
        return self.byname[attributeName].code
```

```

<nameOf>≡
def nameOf(self, attributeNum):
    if isinstance(attributeNum, int):
        return self.bynum[attributeNum].name
    else:
        return attributeNum

```

4.4 Standard RADIUS Definitions

Once created, the object `StandardRadiusDefs` is not expected to change at all. There need be just one such object. Failure to initialize the definitions can be detected and acted upon by checking `StandardRadiusDefs.unread` (which we do in `Attributes.__init__`).

The default filename is “radius-defs.txt”. To read a different file, import and call `intro.setRadiusDefsFilename()` before using or importing anything else from this library.

```

<StandardRadiusDefs>≡
try:
    StandardRadiusDefs = RadiusDefs(intro.StandardRadiusDefsFilename)
except IOError:
    StandardRadiusDefs = RadiusDefs()

```

4.5 Scratch and Dent

[Single/Tuple] Example: An attribute is a tuple of three items: attribute-code, attribute-length, attribute-value. These three are children of attribute. Each one of these three is a single, i.e., no children.

When the size is larger than needed, it is padded to fill the allotted space. How it is padded depends on its type. Integers are filled with leading zeroes. Strings are padded with trailing ascii nuls.

```

<main>≡
if __name__ == "__main__":
    RadiusDefs("radius-defs.txt")
    pass

```

Chapter 5

Client

We expect a single RADIUS client to be interested in talking to multiple RADIUS servers. A channel is a “connection” between a RADIUS client and a RADIUS server. A RADIUS client `Client` is a specialization of network host.

```
<channel.py>≡
# channel.py generated from channel.nw
import socket
from select import select
import time
import thread
import threading
from Queue import Queue
import logging
import sys
import traceback

from intro import RAND_ATTR_NAME
from packet import DissectedPacket, Identifier, Authenticator, \
    PacketError
from attribute import Attributes, StandardRadiusDefs
from radiusTest import RadiusTest
import util

<class Channel>
<main>
# -eof-
```

```
<main>≡  
  if __name__ == "__main__":  
      Channel("radiusServer.osis.wright.edu",  
             "sharedSecret")  
  pass
```


⟨receive⟩
⟨randomPackets⟩

⟨Actions⟩
⟨conductTest⟩

5.1.0.1 Action Customization

The actions (see Section 3.1.1) for the `RadiusTest` objects used by `conductTest` are initialized based on the corresponding instance variables below, if not `None`.

We also keep lists of valid actions so they can be processed en masse without scattering lists of names all over.

```

⟨Actions⟩≡
  ⟨defineActions⟩

  ## Members of [[radiusTest]] (see [[conductTest()]])
  preTest = None
  postTest = None
  postDelayAction = None
  chooseAttr = None
  genCount = None
  genDelay = None
  testActions = ("preTest",
                 "postTest",
                 "postDelayAction",
                 "chooseAttr",
                 "genCount",
                 "genDelay"
                 )

  ## Other members of [[Channel]]s
  # The [[Identifier]] and [[Authenticator]] objects can be
  # anonymous; we only need to call their [[gen()]] methods.
  defaultAttributeNamesGen = []
  attributeNamesGen = defaultAttributeNamesGen
  defaultIdentifierGen = Identifier().gen
  defaultAuthenticatorGen = Authenticator().gen
  identifierGen = defaultIdentifierGen
  authenticatorGen = defaultAuthenticatorGen
  otherActions = ("attributeNamesGen",
                  "identifierGen",
                  "authenticatorGen",
                  )

  ## Members of [[RadiusPacket]]s
  packetLengthGen = None
  pktbytesManipulation = None
  packetAttrsGen = None
  preSend = None
  postSend = None

```

```
packetActions = ("packetLengthGen",
                 "pktbytesManipulation",
                 "preSend",
                 "postSend",
                 "packetAttrsGen",
                 )

# TBD: possible optimizations? 1. Move the lists to the module, so
# they don't get created per-object. 2. Keep a list of all actions
# with them.
#
# 2. is probably worth it right away. 1. means splitting up the code
# in the noweb.
```

`defineActions` is a generalized way of redefining the several types of actions. Its `actionsDict` parameter is a dictionary whose values are functions.

If the key is an attribute name or number, the value is used the value generator for that attribute.

If the key is one of the following, the value is used for the action with that name and setup appropriately. TBD{ Collect ref's to the explanations of each. }

`identiferGen()`

`authenticatorGen()`

`attributeNamesGen()` gives a list of names of attributes to be generated and added to each packet for this channel (see `packetAttrsGen`).

`preTest()`

`postTest()`

`postDelayAction(radiusTest)`

`chooseAttr()`

`genCount()`

`genDelay()`

`prePack(packet)` invoked by `Packet.udpPayload()`. Use to encode User-Password, etc.

`preSend(packet)` invoked by `ChannelPacket.udpPayload()`.

`postSend(packet)` invoked by `ChannelPacket.udpPayload()`.

`packetAttrsGen(packet)` invoked during `ChannelPacket` initialization to generate a list of (attribute name, value) pairs, where the names are from `attributeNamesGen`.

`packetLengthGen(attributeBytes)`

`pktbytesManipulation(pktbytes)`

`<defineActions>≡`

```
def defineActions(self, actionsDict):
    actions = self.testActions + self.otherActions + self.packetActions
    for name, func in actionsDict.items():
        if name in actions:
            setattr(self, name, func)
        elif name in StandardRadiusDefs.byname:
            code = StandardRadiusDefs.numOf(name)
            self.attributes.gen[code] = func
```

```
        else:
            raise ChannelError("Unknown action %s", name)
    pass
```

5.1.1 RADIUS packet Construction

$\langle randomPackets \rangle \equiv$
TBD

5.1.2 RADIUS Communication

`send` is non-blocking; `recv` is blocking.

`send` retries `self.retries` attempts. `recv` times out after `self.timeout` seconds and returns `None` instead of a `DissectedPacket`. (The raw communication functions both raise `ChannelErrors` for these failures, which appear as warnings in the debug output.) The default timeout is 35 seconds, slightly higher than `FreeRADIUS`'s default `max_request_time` of 30 seconds.

A `Channel` instance has two threads. All its `send` methods happen in one thread, and all its `recv` methods happen in a second thread, and both threads are separate from that of the channels owner. The `send` and `recv` should not be in the same thread because we do not want strict mutual exclusion of these. On the extravagant side, we could spawn a new thread for each `send` and make it wait for corresponding `recv`. But for proper testing of RADIUS client, we must not assume that a single `send` will only receive single reply. As a result, the `send` thread in a channel is forever ready to send, and the receive-thread is forever ready to receive any response for the RADIUS server.

If an unanticipated exception is raised in either worker thread, we send the stack trace in an debugging error message and start over rather than stop the thread.

```
<send>≡
def send(self, radiusPkt):
    self.last = radiusPkt
    self.sendQueue.put(radiusPkt)
    pass

def sendRaw(self, rawPkt, port):
    for attempt in xrange(self.retries):
        try:
            self.socket.sendto(rawPkt, (self.server, port))
            return
        except socket.gaierror, e:
            if attempt == (self.retries - 1):
                raise ChannelError("Sending to %s:%s, %s"
                                    % (self.server, str(port), e[1]))
    pass

def sendWorker(self):
    while True:
        p = self.sendQueue.get(True)      # blocking get
        if p.cmd in ("tqm", "trm", "tsm"):
            port = self.acctport
        else:
            port = self.authport
        try:
```

```
        self.sendRaw(p.udpPayload(), port)
    except ChannelError, e:
        self.log.warning(e)
    except PacketError, e:
        self.log.error(e)
    except:
        traceText = "".join(apply(traceback.format_exception,
                                   sys.exc_info()))
        self.log.error("sending over %s:\n" % self
                       + traceText)
pass
```

`recv` is a blocking call. Note that there is a need for tracking the packet source IP address to correctly match replies.

A `DissectedPacket` will be returned if one can be received. If an error of any type occurs (including an entirely unreachable server), `None` will be returned instead once `recv` times out.

```

<receive>≡
def recv(self):
    # signal worker thread
    self.recvEvent.set()
    # block until the packet is ready
    r = self.recvQueue.get()
    return r

def recvRaw(self):
    now = time.time()
    waitto = now + self.timeout
    while now < waitto:
        ready = select([self.socket], [], [], (waitto - now))
        if ready[0]:
            return self.socket.recvfrom(4096)
        else:
            now = time.time()
            continue
        pass
    raise ChannelError("Timeout: no reply from %s after %i seconds"
                       % (self.server, self.timeout))

def recvWorker(self):
    while True:
        # block until a signal from recv()
        self.recvEvent.wait()
        self.recvEvent.clear()
        try:
            (rawreply, source) = self.recvRaw()
            # FIXME: dest IP
            dest = self.socket.getsockname()
            p = DissectedPacket(self.sharedSecret,
                               rawreply,
                               sourceIP=source[0],
                               sourcePort=source[1],
                               destIP=dest[0],
                               destPort=dest[1])

            self.recvQueue.put(p)
        except ChannelError, e:
            self.log.warning(e)

```

```

        self.recvQueue.put(None)
    except PacketError, e:
        self.log.error(e)
    except:
        traceText = "".join(apply(traceback.format_exception,
                                   sys.exc_info()))
        self.log.error("receiving over %s:\n" % self
                       + traceText)
        self.recvQueue.put(None)
pass

```

Central handling of messages from the worker threads.

For now everything goes to stderr. We can trivially save messages in a file instead.

TBD{ FIXME: Each log message is getting logged multiple times for some reason. And not a random number of times—the number is equal to the number of times through `recv()`. E.G. first `recv`: message logged and printed once, second `recv`: message logged and printed twice. third `recv`: no message logged, fourth `recv`: message logged and printed four times. Note that this happens even for log calls in `recv`, not `recvWorker`, so it's not because the message comes from another thread. I can't make it happen in a test program that only does logging and threads, though. I've checked, they are only sent to the logger once (a regular print on the line before a log call always prints once). }

<Handle communication errors>≡

```

# Logging (of events, not packets)
self.log = logging.getLogger("RadiusPktGen Channel for "
                             + self.server)

# handler = logging.FileHandler('some/file')
handler = logging.StreamHandler() # default to sys.stderr
formatter = logging.Formatter("%(asctime)s %(levelname)s %(message)s")
handler.setFormatter(formatter)
self.log.addHandler(handler)

```

<conductTest>≡

```

def conductTest(self, lst=None):
    t = RadiusTest(self)
    t.conductTest(lst)
pass

```

Chapter 6

RADIUS Packets

In this design, we will be referring to three levels of packets: (i) the IP packet that carries the (ii) UDP packet that carries the (iii) RADIUS packet. This design assumes that IP and UDP packets are valid. The invalidity, deliberate or otherwise, is only in the context of RADIUS packets.

```
<packet.py>≡
# packet.py generated from packet.nw
from copy import deepcopy
import time
import struct

from intro import RAND_ATTR_NAME
from attribute import StandardRadiusDefs
from attribute import Attribute, packAttributes, dissectAttributes
import attributeGen
import util
from util import random

<class Identifier>
<class Authenticator>
class PacketError(util.RadiusException): pass
class UnknownPacketError(PacketError): pass

<class RadiusPacket>

<Tests and Demo>

# -eof-
```

6.1 RADIUS Packets

`RadiusPacket` is the a general-use base class representing the RADIUS packet fields. Usually they are actually objects of a subclass suited to a specific role.

```
<class RadiusPacket>≡  
class RadiusPacket(object):  
    <Packet-Type Command>  
    <RadiusPacket.init>  
    <Actions>  
    <Methods of RadiusPacket>  
    <Utility Methods of RadiusPacket>  
    <Packet Composition and Generation Methods of RadiusPacket>  
<Creating RADIUS packets>  
  
<Packet-Type Code/Command Conversion>
```

6.1.1 RADIUS Packet Initialization

`self.command` is the Packet-Type code (the name was chosen to avoid confusion with other uses of the word “code” and “type”). Valid commands are the same abbreviations used in the definitions file. TBD{ List here? Move this explanation to packet.nw? } TBD{ How well are non- Access-Request commands working? }

`self.attributes` is a list of `Attribute` objects. This is a *list* because the order of attributes in a RADIUS packet matters. It may be initialized by an `attrNameValueList`, which will be passed to `compose`.

If a packet object has been identified with a source host, `self.sourceIP` is its IP address as string and `self.sourcePort` is a number. Otherwise, both are `None`. Analogous `self.destIP` and `self.destPort` fields may also be present.

```

<RadiusPacket.init>≡
def __init__(self, command, attrNameValueList=[],
             identifier=0, authenticator="", sharedSecret=""):
    ## Data Members
    self.cmd = command
    self.channel = None
    self.sourceIP = None
    self.sourcePort = None
    self.destIP = None
    self.destPort = None
    self.timestamp = None
    self.identifier = identifier
    self.authenticator = authenticator
    self.sharedSecret = sharedSecret

    self.attributes = []
    self.compose(attrNameValueList)

pass

```

Default actions are provided; the user may substitute new methods with the same parameter list.

The default pre-pack action does encryption of the generated User-Password. Override it to send raw strings instead.

(Actions)≡

```

def defaultPacketLengthGen(self, attributeBytes):
    if attributeBytes is None:
        nb = 0
    else:
        nb = len(attributeBytes)
    return 20 + nb
packetLengthGen = defaultPacketLengthGen

def defaultPktbytesManipulation(self, pktbytes):
    return pktbytes
pktbytesManipulation = defaultPktbytesManipulation

packetAttrsGen = None

def defaultPrePack(_, self):
    for attr in self.attributes:
        if attr.code == StandardRadiusDefs["User-Password"].code:
            attr.value = attributeGen.md5Hidden(attr.value,
                                                self.sharedSecret,
                                                self.authenticator)
    pass
prePack = defaultPrePack

```

Pre- and post-sending actions should not be run unless the packet is being sent, so they are not run for non-channel packets. `packetAttrsGen` also only makes sense for packets being generated and sent, so it requires a `Channel`.

```

<Actions Specific to ChannelPacket>≡
def defaultPreSend(_, self):
    self.timestamp = time.time()
    print("Packet to be sent:")
    print(self)
    pass
preSend = defaultPreSend

postSend = None

def defaultPacketAttrsGen(_, self):
    codes = [ StandardRadiusDefs.numOf(name)
              # Given names, but Attribute objects for self.attributes
              # are created by code.
              for name in util.invoke(self.channel.attributeNamesGen) ]
    attrs = [ Attribute(*pair)
              for pair in
                self.channel.attributes.generateSpecificAttributes(codes) ]
    self.attributes += attrs
    pass
packetAttrsGen = defaultPacketAttrsGen

```

There are three general situations in which packets are created:

- For use over a `Channel` (`ChannelPacket`).
- From raw data (`DissectedPacket`). The data in question may come directly from the network, or from files in libpcap format.
- From existing `RadiusPackets`. For replies, use `generateReply` and `generateFollowUpAQM`. TBD{ Make those work }. TBD{ There should also be general “deep copy the packet, create a new ID, etc.” method. }

```

<Creating RADIUS packets>≡
<class ChannelPacket>
<class DissectedPacket>

```

Packets being *sent* should have a channel to query for their shared secret, attribute generators, actions, identifier and authenticator. To override the channel's authenticator and identifier generators for a specific packet, specify a non-None argument to the constructor.

```

<class ChannelPacket>≡
class ChannelPacket(RadiusPacket):
    def __init__(self, command, channel, attrNameValueList=[],
                 identifier=None, authenticator=None):
        if identifier is None:
            identifier = channel.identifierGen()
        else:
            identifier = util.invoke(identifier)
            try:
                identifier = int(identifier)
            except ValueError:
                raise PacketError("Bad identifier %s" % identifier)
            pass
        if authenticator is None:
            authenticator = channel.authenticatorGen()
        else:
            authenticator = util.invoke(authenticator)
            pass
        superclass = super(self.__class__, self)
        superclass.__init__(command, attrNameValueList,
                            # Other parameters from channel
                            identifier=identifier,
                            authenticator=authenticator,
                            sharedSecret=channel.sharedSecret)

        self.channel = channel
        # Custom actions from the channel, if any
        for action in channel.packetActions:
            func = getattr(channel, action)
            if func is not None:
                setattr(self, action, func)
        # If the channel says some A/V pairs are to be generated, do
        # so (possibly overriding const pairs from aNVL param.)
        util.invoke(self.packetAttrsGen, self)

        pass

    <Actions Specific to ChannelPacket>
    <ChannelPacket.udpPayload>
    pass

```

`DissectedPackets` are built from byte sequences (i.e. received from outside the program).

We deliberately do not verify the integrity of the attributes. We do not assume that `pktbytes` is well-formed. We assume only that it is a list of bytes. However, if the byte sequence is too short to contain the necessary information (less than 20 bytes), a `PacketError` is raised.

`self.timestamp` is a (possibly fractional) time in seconds since the epoch, or `None` if it is not defined for a packet.

```

<class DissectedPacket>≡
class DissectedPacket(RadiusPacket):
    def __init__(self, sharedSecret="", pktbytes="",
                 sourceIP=None, sourcePort=None,
                 destIP=None, destPort=None,
                 timestamp=None):
        length = len(pktbytes)
        if (length < 20):
            raise PacketError("Invalid pktbytes: %i < 20, too short"
                               % length)

        code = ord(pktbytes[0])
        try:
            cmd = typeCodeToCommand[code]
        except KeyError:
            raise UnknownPacketError("Unsupported RADIUS code %i"
                                      % code)

        superclass = super(self.__class__, self)
        superclass.__init__(cmd, [],
                            sharedSecret=sharedSecret)

        self.sourceIP = sourceIP
        self.sourcePort = sourcePort
        self.destIP = destIP
        self.destPort = destPort
        (self.identifier,
         self.length,
         self.authenticator) = struct.unpack("!BH16s", pktbytes[1:20])
        self.attributes = dissectAttributes(pktbytes[20:])
        self.timestamp = timestamp
        pass
pass

```

6.1.2 Composition and Generation of Packets

<Packet Composition and Generation Methods of RadiusPacket>≡
<compose>
<readPacketDumpFile>
<generateReply>
<generateFollowUpAQM>

`compose` is the method to set attributes when providing a constant value.

We deliberately do not perform validity checks on its parameters because it is useful to generate illegal packets for test purposes. TBD However, we should check for type validity. It returns a well-formed RADIUS packet.

When the special attribute name `channel.RAND_ATTR_NAME` is encountered, a random attribute name is generated and substituted. This allows an attribute name/value pair list to produce different attribute names (but with the same pattern of values) by passing it to `compose` multiple times.

```
<compose>≡
def compose(self, attrNameValueList):
    for name, value in attrNameValueList:
        if name == RAND_ATTR_NAME:
            name = random.choice(StandardRadiusDefs.byname.keys())
            pass
        attr = StandardRadiusDefs[name]
        self.attributes.append(Attribute(attr.code,
                                         util.invoke(value)))
    pass
```

6.1.2.1 From Raw RADIUS packets

TBD{ }

```
<readPacketDumpFile>≡
def readPacketDumpFile(self, filename):
    while True:
        yield "one radius packet"
    pass
```

6.1.2.2 Response RADIUS packets

`generateReply` is intended for RADIUS server use.

`accept` is a boolean meaning “reject” if False.

```

<generateReply>≡
def generateReply(self, requestMessage, accept):
    TBD()
    pass

```

`generateFollowUpAQM` is for the RADIUS client that intends to respond to the challenge it received.

```

<generateFollowUpAQM>≡
def generateFollowUpAQM(self, previousRequestMessage, replyMessage):
    pass

```

6.1.3 Methods of RadiusPacket

```

<Methods of RadiusPacket>≡

def isWellFormed(self):
    # FIXME set in subclasses
    return False

def isTemporallyValid(self):
    # FIXME implementation TBD
    return False

def isRadiusReply(self, sent, recd):
    """ sent and recd are [] of bytes. """
    if (sent[1] != recd[1]) or (sent[0] != AccessRequest):
        return False
    # TBD needs to match UDP src/dst ports also.
    # see response-authenticator in ERFC2865 server.tex
    shouldbe = MD5(recd[0:4] + sent[4:20] +
                   recd[20:] + self.sharedSecret)
    return shouldbe == recd[4:20]

```

6.1.3.1 Utility Methods of RadiusPacket

`isValid` raises a `PacketError` if the attribute/value pairs of the packet do not satisfy the defined relations between packet type (`self.cmd`) and number of values for an attribute.

(RadiusPacket.isValid)≡

```
def isValid(self):
    try:
        packetType = typeCommandToCode[self.cmd]
    except KeyError:
        raise PacketError("Unknown packet type: %s" % self.cmd)

    counts = {}
    for attr in self.attributes:
        code = attr.code
        if counts.has_key(code):
            counts[code] += 1
        else:
            counts[code] = 1

    for adef in StandardRadiusDefs:
        # Loop over all known attributes to ensure all w/=1 are present
        code = adef.code
        constraint = getattr(adef, self.cmd)
        if (constraint == "==0") and (counts[code] != 0):
            raise PacketError("%s packets must not have %s"
                               % (self.cmd, adef.name))
        elif (constraint == "==1") and (counts[code] != 1):
            raise PacketError("%s packet must have %s attribute"
                               % (self.cmd, adef.name))
        elif (constraint == "<=1") and (counts[code] > 1):
            raise PacketError("%s packet has >1 %s attributes"
                               % (self.cmd, adef.name))

    # >=
    pass
```

```

<Utility Methods of RadiusPacket>≡
def __str__(self):
    authr = self.authenticator.encode("string_escape")
    if self.timestamp is not None:
        timestamp = "\tTimestamp=%s\n" % time.ctime(self.timestamp)
    else:
        timestamp = ""
        pass
    s1 = ["Radius %s id=%d\n" % (self.cmd,
                               self.identifier),
         "\tAuthenticator=%s\n" % authr,
         "\tShared Secret=%s\n" % self.sharedSecret,
         "%s" % timestamp,      # time may not be present
         "\n"]
    # Append the list of printed attributes/value pairs
    s1[-1:-1] = [ "\t%s\n" % str(attr)
                  for attr in self.attributes ]
    return "".join(s1)

```

<RadiusPacket.udpPayload>

<RadiusPacket.isValid>

<Packet-Type Command>

`udpPayload` assembles the packet into a byte sequence suitable for sending over the network.

Validity checking not done by default; the `validate` parameter turns it on. If checks are requested and fail, a `PacketError` is raised (see `RadiusPacket.isValid()`, page 58).

The `prePack` action is invoked before assembling anything, and can thus be used to modify the data structures of the packet. The `self.pktbytesManipulation` action (see page 43) is invoked after the payload is assembled.

```

<RadiusPacket.udpPayload>≡
def udpPayload(self, validate=False):
    if validate:
        self.isValid()

    util.invoke(self.prePack, self)
    attributeBytes = packAttributes(self.attributes)
    header = struct.pack("!BBH16s",
                        typeCommandToCode[self.cmd],
                        self.identifier,
                        self.packetLengthGen(attributeBytes),
                        self.authenticator)

    pktbytes = header + attributeBytes
    pktbytes = util.invoke(self.pktbytesManipulation, pktbytes)
    if not isinstance(pktbytes, str):
        raise PacketError("pktbytesManipulation action %s"
                          " did not return a byte sequence."
                          % self.pktbytesManipulation)

    return pktbytes

```

Assembling a `ChannelPacket`'s payload is assumed to mean sending the packet, so additional actions are invoked.

```

<ChannelPacket.udpPayload>≡
def udpPayload(self, validate=False):
    util.invoke(self.preSend, self)
    pktbytes = RadiusPacket.udpPayload(self, validate=validate)
    util.invoke(self.postSend, self)
    return pktbytes

```

Ignore capitalization for packet command codes.

I find that the string makes things much easier to follow.

<Packet-Type Code/Command Conversion>≡

```
typeCommandToCode = { "aqm":1,
                      "aam":2,
                      "ajm":3,
                      "acm":11,
                      }
typeCodeToCommand = { 1:"aqm",
                     2:"aam",
                     3:"ajm",
                     11:"acm",
                     }
```

<Packet-Type Command>≡

```
def get_cmd(self):
    return self._cmd
def set_cmd(self, value):
    self._cmd = value.lower()
cmd = property(get_cmd, set_cmd)
```

6.2 RADIUS Identifier

`IdentifierGen` produces random RADIUS identifiers. Our implementation is guaranteed to not repeat an identifier in a run length of 256. It does not depend on any `self` fields, but we need a “new” identifier generator each time `RadiusPacket()` is invoked. But, being one byte there are only 256 distinct values. TBD{ RFC 2865 says we should use UDP port to disambiguate identifiers. }

`ix`, `values` should be global to the `IdentifierGen`, but specific to each object created – i.e., these are not class variables.

```
<class Identifier>≡
class Identifier:
    def __init__(self):
        self.ix = 0
        self.values = range(0, 256)
        random.shuffle(self.values)
        pass

    def gen(self):
        if self.ix == 256: self.ix = 0
        self.ix += 1
        return self.values[self.ix - 1]

pass
```

6.3 Radius Authenticator

`AuthenticatorGen` is a producer of random RADIUS authenticators, as a 16-byte list.

The RFC requires authenticator values to be unpredictable and temporally unique. For uniqueness, in general, we need to save the authenticators used so far; the length of this list called `sofar` will not exceed `maxLength=1000`. For unpredictability we rely on Python's `random` module.

Notes:

- Currently changing authenticator generation is harder than some customizations. The user can replace the `authenticatorGen` member of `Channel` but not replace `_simpleGen` here.
- Values in `sofar` are in temporal order, latest last. They used to be time stamped, but that wasn't used and makes checking the list more complex, so for now they're not. Another idea is to use a dictionary with authenticators as keys.
- We ensure that each authenticator has not been used within the last 1000 generated in this run of the program. However, if we generated certain 16-bit random numbers frequently, the check-and-retry loop could become long. Another idea is to try e.g. a maximum of 5 times.

```

<class Authenticator>≡
class Authenticator:
    def __init__(self):
        self._sofar = []                # must save old values
        self.maxLength = 1000

    def _simpleGen(self):
        return attributeGen.randomByteSeq(16)

    def gen(self):
        while True:
            v = self._simpleGen()
            if v not in self._sofar:    # check history
                break                  # use only if not found
            self._sofar.append(v)

        if len(self._sofar) > self.maxLength:
            del self._sofar[0]

        return v

pass

```

```
<Tests and Demo>≡
if __name__ == "__main__":
    from channel import Channel
    achannel = Channel(server="server",
                      sharedSecret="shared secret")
    p = ChannelPacket("aqm", achannel,
                      [("User-Name", "username"),
                       ("User-Password", "passwd")
                      ])
    p.identifier = 106
    p.authenticator = "\x34}86y7458yhoj6n\xB3"
    assert(p.udpPayload() ==
           "\x01j\x0004}86y7458yhoj6n\xb3\x01\nusername\x02\x12\xee\xa9\x96n\xeb\xb4\xc2\x")
    print "PASS"
    pass
```

Chapter 7

Packet Utilities

```
<packetUtil.py>≡
# packetUtil.py generated from packetUtil.nw
import struct
import socket
import binascii

from packet import ChannelPacket, DissectedPacket
import util

<Packet dumps>
<General packet generators>

<Lower-Layer Packets>

# -eof-
```

7.1 RADIUS Packet Dumps

We make use of the `pylibpcap` bindings to read the `libpcap` file format use by `tcpdump` and `Ethereal`. See §1.2.4 for an example script using capture files.

This dependency can be made optional (allowing our library to successfully load with all functionality not related to packet dumps) by adding a Python warning filter. One way to do this is invoking Python as:

```
python -W ignore:radiuspktgen.util.PktGenWarning <script>
```

<Packet dumps>≡

```
try:
    import pcap
except ImportError:
    from warnings import warn
    warn("libpcap bindings not found; "
         "packet dumps will not work. "
         "You should install pylibpcap 0.5.1.",
         util.PktGenWarning)
pass
```

```
class DumpException(util.PktGenException): pass
```

```
class RadiusDumpException(DumpException, util.RadiusException): pass
```

<Reading RADIUS packet dumps>

<Writing RADIUS packet dumps>

`RadiusDumpReader` reads a capture file containing RADIUS messages. Currently, all packets within the capture file must be RADIUS messages of types supported by our library. (Ethereal can be used to filter dumps for these packets.)

`self.sharedSecret` is a mapping from (sourceIP, destIP) tuples to strings containing the shared secret. This mechanism is convenient for most cases, but has two caveats:

1. Usually, a duplicate mapping (destIP, sourceIP) must be given for (i.e., when the destination server is acting as the source of replies).
2. This will not work at all if e.g. a host A is a client of B yet B connects to a server on A with a different shared secret.

<Reading RADIUS packet dumps>≡

```
class RadiusDumpReaderException(RadiusDumpException): pass
```

```
class RadiusDumpReader(object):
    def __init__(self, filename, sharedSecretMapping={}):
        self.filename = filename
        self.sharedSecrets = sharedSecretMapping
        pass
```

```
    def open(self):
        cap = pcap.pcapObject()
        cap.open_offline(self.filename)
        return cap
```

<RadiusDumpReader.dissect_frame_udpip>

<Searching RADIUS dump files>

```
pass
```


These methods return a list of all requests in the dump file being read by a RadiusDumpReader for requests with the specified attribute/value pairs, identifier, etc. We also provide a generalized `findIf` method for returning the list of requests for which a given predicate returned true.

All searches may be limited to returning at most `count` requests (though there is no way to limit the number of requests searched but not matched).

(Searching RADIUS dump files)≡

```
def findIf(self, predicate, count=False):
    list = []
    self.count = count

    def callback(pktlen, frame, timestamp):
        if self.count and self.count < 1:
            return False
        decoded = self.dissect_frame_udpip(frame)
        decoded.timestamp = timestamp
        if util.invoke(predicate, decoded):
            list.append(decoded)
            if self.count:
                self.count -= 1
            pass
        pass

    cap = self.open()
    cap.loop(0, callback)
    return list

def byId(self, id, count=False):
    def pred(packet):
        return (packet.identifier == id)
    return self.findIf(pred, count)

def byAuthenticator(self, authenticator, count=False):
    def pred(packet):
        return (packet.authenticator == authenticator)
    return self.findIf(pred, count)

def byCommand(self, command, count=False):
    def pred(packet):
        return (packet.cmd == command)
    return self.findIf(pred, count)
```

```
def byTime(self, before, after, count=False):
    def pred(packet):
        return (packet.timestamp >= before
                and packet.timestamp <= after)
    return self.findIf(pred, count)

def byHost(self,
            sourceIP=False, sourcePort=False,
            destIP=False, destPort=False,
            count=False):
    def pred(packet):
        match = True
        if sourceIP and packet.sourceIP != sourceIP:
            return False
        if sourcePort and packet.sourcePort != sourcePort:
            return False
        if destIP and packet.destIP != destIP:
            return False
        if destPort and packet.destPort != destPort:
            return False
        return match
    return self.findIf(pred, count)

def byAVPairs(self, count=False, *attrNameValueList):
    def pred(packet):
        for name, value in attrNameValueList:
            if packet.attributes[name] != value:
                return False
        pass
        return True
    return self.findIf(pred, count)
```

`RadiusDumpWriter` writes RADIUS messages to a capture file. Appropriate UDP, IP, and Ethernet II headers and footers are generated for this purpose.

The underlying file object is automatically closed when the `RadiusDumpWriter` object is destroyed.

(*Writing RADIUS packet dumps*)≡

```
class DumpWriterError(RadiusDumpException): pass

class DumpWriter(object):
    def __init__(self, filename):
        try:
            dump = file(filename, "w")
        except IOError:
            raise DumpWriterError("Couldn't write to %s." % filename)

        # FIXME: can we bind libpcap writer (without a live interface)
        # instead of duplicating parts? if so, we really should.

        # Headers have the endianness of the capturing machine.
        dump.write(struct.pack("I", 0xa1b2c3d4)) # magic
        dump.write(struct.pack("H", 2))         # major version
        dump.write(struct.pack("H", 4))         # minor version
        dump.write(struct.pack("I", 0))         # FIXME: timezone
        dump.write(struct.pack("I", 0))         # FIXME: "accuracy"?
        dump.write(struct.pack("I", 0))         # FIXME: length of ...?
        dump.write(struct.pack("I", 1))         # for Ethernet

        self.dump = dump
        pass

    def writeData(self, frame, timestamp):
        self.dump.write(struct.pack("I",          # seconds
                                   int(timestamp)))
        self.dump.write(struct.pack("I",          # microseconds
                                   int(timestamp % 1 * 1e6)))
        self.dump.write(struct.pack("I",          # length of portion captured
                                   len(frame)))
        self.dump.write(struct.pack("I",          # length on wire
                                   len(frame)))
        self.dump.write(frame)                  # frame data
        pass

    def __del__(self):
        self.dump.close()
        pass
```

```
pass
```

```
class RadiusDumpWriter(DumpWriter):
    def writePacket(self, packet):
        pktbytes = packet.udpPayload()
        udp = UDPPacket(packet.sourcePort,
                       packet.destPort,
                       network=IP4Packet(packet.sourceIP, packet.destIP))
        self.writeData(udp.assemble(pktbytes), packet.timestamp)
    pass
```

7.2 General Packet Generators

These are general forms for generating packet sequences that follow common patterns. Like the general attribute generators of §9.2, each function creates and returns a closure suitable for use as a generalized value in our library (i.e., one whose successive return values will be the sequence indicated).

⟨General packet generators⟩≡
⟨sequence of channel packets⟩

Generation of packets with (possibly varying) attributes and values also has a general form. Successive invocations of the closure returned by this give a sequence of packets. If `args` includes closures that vary their return values, the sequence will vary.

⟨sequence of channel packets⟩≡

```
def genChannelPackets(command, channel, *args):
    def inner():
        while True:
            yield ChannelPacket(command, channel, *args)
        pass
    return inner().next
```

7.3 Lower-Layer Packets

These classes represent the lower levels of packets (transport, network, and link) carrying our application's packets. Sophisticated manipulation of these layers is not our focus; rather, these are used primarily to read and write raw network captures of the application data.

A packet in a given protocol is represented by an object containing an object of the class for the lower-level protocol in which it is to be encapsulated. For each class there is also a dissector function to initialize an object from a byte sequence.

Notice that these packet classes each have an `assemble()` method taking the sequence of data bytes. This means that objects of these classes can be used as generators of similar packets with different payloads by repeatedly assembling the same object with varying data.

```

<Lower-Layer Packets>≡
  <Packet Utilities>
  <class EtherFrame>
  <class IP4Packet>
  <class UDPPacket>
  <Known Protocols>

```

```

<Packet Utilities>≡
def checksum(bytes):
    if len(bytes) % 2 == 1:
        bytes += "\0"
    pass
    cs = 0
    for i in xrange(len(bytes)/2):
        cs += struct.unpack("!H", bytes[2*i:2*i+2])[0]
    pass
    cs = (cs >> 16) + (cs & 0xffff)
    return struct.pack("!H", ~(cs + (cs >> 16)) & 0xffff)

class PacketFormatError(util.PktGenException): pass

```

After reading a protocol type header, a lower layer object can look up its dissector from these known protocols.

```

<Known Protocols>≡
class ProtocolTypeError(util.PktGenException): pass
networkTypes = {0x0800: dissectIP4Packet}
transportTypes = {socket.IPPROTO_UDP: dissectUDPPacket}

```

```

<class UDPPacket>≡
class UDPPacket(object):
    def __init__(self,
                 sourcePort=0,
                 destPort=0,
                 dataBytes=None,
                 network=IP4Packet(protocol=socket.IPPROTO_UDP)):
    if network.protocol is None:
        network.protocol = socket.IPPROTO_UDP
    pass
    self.network = network

    self.sourcePort = sourcePort
    self.destPort = destPort
    self.dataBytes = dataBytes
    pass

    def assemble(self, dataBytes=None):
    if dataBytes is None:
        dataBytes = self.dataBytes
    if dataBytes is None:
        raise PacketFormatError("No data bytes to assemble")

    udpLength = struct.pack("H", 8 + len(dataBytes))
    udpBytes = (struct.pack("!HH", self.sourcePort, self.destPort)
               + udpLength)
    sourceIP = util.packIPAddress(self.network.sourceIP)
    destIP = util.packIPAddress(self.network.destIP)
    udpBytes += checksum(sourceIP + destIP
                       + struct.pack("!xB", socket.IPPROTO_UDP)
                       + udpLength + udpBytes
                       + dataBytes)
    return self.network.assemble(udpBytes + dataBytes)

    pass

    def dissectUDPPacket(network, udpBytes):
    if len(udpBytes) < 8:
        raise PacketFormatError("UDP packet too short")
    (sourcePort,
     destPort,
     dataLen) = struct.unpack("!HHH",
                              udpBytes[0:6])
    return UDPPacket(network=network,
                    sourcePort=sourcePort,

```

```
destPort=destPort,  
dataBytes=udpBytes[8:])
```

```

<class IP4Packet>≡
class IP4Packet(EtherFrame):
    def __init__(self,
                 sourceIP="0.0.0.0",
                 destIP="0.0.0.0",
                 protocol=None,
                 frame=EtherFrame(ethertype=0x0800)):
        if frame.ethertype is None:
            # FIXME: also MAC addresses for this?
            frame.ethertype = 0x0800
            pass
        self.frame = frame

        self.sourceIP = sourceIP
        self.destIP = destIP
        self.protocol = protocol
        pass

    def assemble(self, dataBytes):
        ipBytes = (struct.pack("!BBHHBBxx",
                               (4 << 4),           # ver
                               | (5 & 0x0f),        # IHL always 5
                               0,                  # TOS
                               20 + len(dataBytes), # total length
                               1,                  # ident
                               0,                  # flags, fragment offset
                               33,                 # ttl
                               self.protocol)
                  + util.packIPAddress(self.sourceIP)
                  + util.packIPAddress(self.destIP))
        ipBytes = ipBytes[:10] + checksum(ipBytes) + ipBytes[12:]
        return self.frame.assemble(ipBytes + dataBytes)

    pass

    def dissectIP4Packet(frame, ipBytes):
        if len(ipBytes) < 20:
            raise PacketFormatError("IP packet too short")
        protocol = ord(ipBytes[9])
        try:
            transportDissector = transportTypes[protocol]
        except KeyError:
            raise ProtocolTypeError("Unknown transport protocol %s"
                                     % protocol)

        ver_ihl = ord(ipBytes[0])

```

```

version = (ver_ihl & 0xf0) >> 4
if version != 4:
    raise ProtocolTypeError("Only IPv4 is currently supported")
ihl = ver_ihl & 0x0f
totalLen = socket.ntohs(struct.unpack("!H", ipBytes[2:4])[0])
ip = IP4Packet(frame=frame,
               sourceIP=util.unpackIPAddress(ipBytes[12:16]),
               destIP=util.unpackIPAddress(ipBytes[16:20]),
               protocol=protocol)
return transportDissector(ip, ipBytes[ihl * 4:])

```

<class EtherFrame>≡

```

class EtherFrame(object):
    def __init__(self,
                 destMAC="00:00:00:00:00:00",
                 sourceMAC="00:00:00:00:00:00",
                 ethertype=None):
        self.destMAC = destMAC
        self.sourceMAC = sourceMAC
        self.ethertype = ethertype
        pass

    def assemble(self, dataBytes):
        frame = (util.packMACAddress(self.destMAC)
                 + util.packMACAddress(self.sourceMAC)
                 + struct.pack("!H", self.ethertype)
                 + dataBytes)
        frame += struct.pack("i", binascii.crc32(frame))
        return frame

    pass

def dissectEtherFrame(frameBytes):
    try:
        ethertype = struct.unpack("!H", frameBytes[12:14])[0]
        netDissector = networkTypes[ethertype]
    except KeyError:
        raise ProtocolTypeError("Unknown network protocol type %s"
                                % ethertype)
    frame = EtherFrame(destMAC=util.unpackMACAddress(frameBytes[0:6]),
                       sourceMAC=util.unpackMACAddress(frameBytes[6:12]),
                       ethertype=ethertype)
    return netDissector(frame, frameBytes[14:])

```

Chapter 8

Attributes

The file `attribute.py` captures RFC 2865 definition of attributes into Python objects (of class `AttributeDef`). The `Attributes` class is a collection of functions for working with the known attributes. An attribute is conceptually a triplet (code, length, value) along with information about where and how many times it is allowed to occur.

A Channel object has a `Attributes` object as a component. Currently, this set of attribute definitions and generators is used for all packets created by the channel; the expectation is that multiple packets will be generated for each `Attributes` setup.

```
<attribute.py>≡
# attribute.py generated from attribute.nw
import struct

from defs import StandardRadiusDefs
import util
from util import random

class AttributeError(util.RadiusException): pass
class MissingAttributeError(AttributeError): pass

class Attributes:
    <Attributes.init>
    <Methods of Attributes>
    pass
<dissectAttributes>

<class Attribute>

<Tests and Demo>
```

```
# -eof-
```

8.1 Known Attributes

The data fields of class `Attributes` are:

¹

[`self.gen:`] A mapping of attribute numbers to parameterless functions. Each *i*-th function should return a value for attribute numbered *i*. The code is *i*, and the attribute `length` field is “supplied” by this `Attributes` class. It is expected that one or more of these generating functions are replaced by custom generators. The default generators initialized generate random but legitimate values.

[`self.isa:`] is also a mapping of attribute numbers to functions. Each *i*-th should be a boolean function that can verify that its argument is a allowable value for attribute numbered *i*. These functions have two parameters: the length, and attribute value. As above, the values of this list may be altered.

A good example is the combination of the `default_chooseAttr` test action and `randomPacketTriplets` method in `RadiusTest`. Unless the interface changes.
TBD{ Remove? }

⟨Methods of Attributes⟩≡
⟨Choosing Attributes and Values⟩

¹Ben: “should” vs “must”?

We expect the class `Attributes` to be instantiated with a file name just once. All other instantiations are without a file name, and we expect these objects to alter the `gen` list. The `len(self.gen) == 256` so that any one byte attribute code will have a function.

If `StandardRadiusDefs` (see above) could not be read, this will raise a `MissingAttributeError`.

```

<Attributes.init>≡
def __init__(self):
    global StandardRadiusDefs
    if StandardRadiusDefs.unread:
        raise MissingAttributeError("Standard RADIUS definitions not found.")

    # copy the standard functions for possible alteration
    self.gen = list(StandardRadiusDefs.gen)
    self.isa = list(StandardRadiusDefs.isa)

```

In `composeAttribute`, we deliberately omit validity checks such as the length of the value is ≤ 254 . It returns a byte sequence.

```

<composeAttribute>≡
def composeAttribute(self, code, value, length=None):
    if length is None: length = len(value)
    self.att += [(code, length, value)]
    pass

def composeAttributeFromNameValue(self, nmva):
    return composeAttribute(self, attributeNumber(nmva[0]), nmva[1])

def composeAttributesFromNameValueList(self, nmvalst):
    map(composeAttribute, nmvalst)
    pass

```

8.2 Storing Attributes

`Attribute` objects represent an instance of an attribute with a value. The class provides methods for working with that instance.

```
<class Attribute>≡
class Attribute(object):
    def __init__(self, code, value, length=None):
        if length is not None:
            self.autoLength = False
            self.length = length

        self.code = code
        self.value = value
        pass

<Attribute pretty-printing>
<Attribute.value>
<Attribute.pack>

<packAttributes>
```

The `Attribute.value` properly handles values of various types and lengths. Values may be Python ints or objects convertible to strings. We raise an `AttributeError` if the type of `value` cannot be converted to either match the definition for the given code or be used as an arbitrary byte sequence. Validity and format checking is deliberately minimal to allow use of invalid values.

The length is by default determined from the value and type using the equality `length == len(value) + 2`. Because we permit the deliberate generation of invalid packets, however, an overriding length may be specified at initialization. If this is done, automatic lengths are disabled for further changes to this object (for example, if `autoLength` is `False` for a “User-Password”, the length will *not* be properly recalculated when it is encrypted).

```

<Attribute.value>≡
    autoLength = True

def get_value(self):
    return self._value

def realLength(self):
    if isinstance(self.value, int):
        return 4 + 2
    else:
        return len(self.value) + 2

def set_value(self, value):
    if not (hasattr(value, "__str__") or isinstance(value, int)):
        raise AttributeError("Can't convert value for %s to"
                               "byte string"
                               % StandardRadiusDefs[self.code].name)

    type = StandardRadiusDefs[self.code].type
    if type == "uint":
        try:
            value = int(value)
        except ValueError:
            try:
                value = int((struct.unpack("!I", value))[0])
            except:
                pass
        except TypeError:
            raise AttributeError("Can't convert value for %s to int"
                                   % StandardRadiusDefs[self.code].name)

    elif type == "ipaddr":
        if isinstance(value, int):
            value = struct.pack("!I", value)

```

```
        else:
            value = str(value)
            if len(value) >= 7 and value.count(".") == 3:
                # Given a formatted IP address, must be converted
                value = util.packIPAddress(value)
                pass
            pass
        pass

    else:
        value = str(value)

    self._value = value
    if self.autoLength:
        self.length = self.realLength()

value = property(get_value, set_value)
```

pack this `Attribute` into a byte sequence.

This relies on `set_value` to set `self.value` to either a Python int or some object that can be converted to a `str`. In the string case, that value will be used directly as the value's byte sequence (this function does no validity or format checking).

```

<Attribute.pack>≡
def pack(self):
    if isinstance(self.value, int):
        evalue = struct.pack("!I", self.value)
    else:
        # A string encodes to itself
        # (and set_value ensures we have a string)
        evalue = self.value

    try:
        return(struct.pack("!BB", self.code, self.length)
               + evalue)
    except struct.error, e:
        if not isinstance(self.code, int):
            prop = "code"
        elif not isinstance(self.length, int):
            prop = "length"
        else:
            raise AttributeError("Unanticipated Attribute.pack"
                                "error for %s" % self)
        raise AttributeError("Attribute.code isn't an int for %s"
                              % self)
    except TypeError, e:
        # concatenation failed above, need to add an if type(): pack
        raise AttributeError("Unanticipated %s case in Attribute.pack"
                              % type(evalue))

```

`packAttributes` converts a list of `Attribute` objects (as stored by `RadiusPacket`) into a byte sequence suitable for sending over the network.

```

<packAttributes>≡
def packAttributes(list):
    packed = [ attr.pack() for attr in list]
    return "".join(packed)

```

`dissectAttributes` builds a list of `Attribute` objects from a byte sequence. If there is an error, it raises an `AttributeError`.

Possible errors are:

- “Way too short”, when the byte sequence length is 1.
- “Not long enough”, when the actual length of the byte sequence given is less than the attribute lengths specified in the sequence.

$\langle dissectAttributes \rangle \equiv$

```
def dissectAttributes(bytes):
    successCode = 0
    lst = []
    while True:
        length = len(bytes)
        if length == 0:                # done, good
            break
        if length == 1:
            raise AttributeError("Way too short")
        attrlen = ord(bytes[1])        # OK to have attrlen == 2
        if length < attrlen:
            raise AttributeError("Not long enough")

        lst.append(Attribute(ord(bytes[0]),
                             bytes[2:attrlen],
                             attrlen))
        bytes = bytes[attrlen:]
    pass
    return lst
```

We handle several things specially when printing:

IP Addresses Original dotted quad printed instead of the raw byte string.

Potentially invalid lengths When generating invalid packets it may be useful to know just how invalid they are.

```

<Attribute pretty-printing>≡
def __str__(self):
    pvalue = self.value
    attr_def = StandardRadiusDefs[self.code]
    if attr_def.type == "ipaddr" and len(pvalue) == 4:
        pvalue = util.unpackIPAddress(pvalue)
    else:
        pvalue = str(pvalue).encode("string_escape")
    pass

    if self.autoLength:
        length_str = "%i" % self.length
    else:
        length_str = ("%i (really %i)"
                      % (self.length, self.realLength()))
    return("attr %s: len=%s, value=%s"
          % (attr_def.name, length_str, pvalue))

```

8.3 Choosing Attributes and Values

```

<Choosing Attributes and Values>≡
  <Generate Specific Attributes>
  <Generate Random Attributes>

```

The functions below generate values for specific attribute(s). These are random values if the `self.gen` is unchanged from what was assigned initially. An “attribute designator” is the name or number by which an attribute is identified in `StandardRadiusDefs`.

`generate` returns one value for the attribute with the given designator. It uses `util.invoke` to call the generator, so if a constant value is provided instead of a function, that value is always returned.

`generateSpecificAttributes` maps a list of attribute designators to a list of (attribute designator, generated value) pairs. Note that attribute numbers may be duplicated. The order in which these numbers are listed must not be altered. The RFC has incomplete specifications regarding that.

TBD{ Above order remark? }

⟨Generate Specific Attributes⟩≡

```
def generate(self, attr):
    if not StandardRadiusDefs.has_key(attr):
        raise MissingAttributeError("Unknown attribute %s" % attr)
    if not isinstance(attr, int):
        attr = StandardRadiusDefs.numOf(attr)
    return util.invoke(self.gen[attr])

def generateSpecificAttributes(self, attrs):
    return [ (attr, self.generate(attr)) for attr in attrs ]
```

These methods randomly select attributes to generate.

Attributes are chosen from the given `names`, or all defined attributes if no names are given.

Unless `uniquify` is true, the list of generated attributes may contain repetition.

TBD{ Select attributes from the supplied names appropriate for message type.
}

⟨Generate Random Attributes⟩≡

```
def generateRandomAttributes(self, count,
                             names=StandardRadiusDefs.byname.keys(),
                             messageType=None,
                             uniquify=False):
    names = self.generateRandomAttributeNameNames(count,
                                                    names,
                                                    messageType,
                                                    uniquify)
    return self.generateSpecificAttributes(names)

def generateRandomAttributeNameNames(self, count,
                                     names=StandardRadiusDefs.byname.keys(),
                                     messageType=None,
                                     uniquify=False):
    return util.randomSample(names, count, uniquify=uniquify)
```


Chapter 9

Attribute Generation

Attribute value generation is straightforward except for a few that involve MD5-related computations.

```
<attributeGen.py>≡  
# attributeGen.py generated from attributeGen.nw  
  
import string  
import md5  
import util  
from util import random  
  
<Basic Type Generators>  
<MD5-hidden>  
<MD5-exposed>  
<Generators for each attribute>  
<General Value Generators>
```

9.1 Basic Type Generators

```
<Basic Type Generators>≡  
<randomByte>  
<randomUInt32>  
<randomIpAddress>  
<randomMacAddress>
```

`randomByte` returns a random byte as an `int`, optionally in the range `a]`, `[b`.

```
<randomByte>≡
def randomByte(a=0, b=255):
    return util.randint(a, b)
```

`randomUInt32` returns a random unsigned integer of 32 bits, optionally in the range `a]`, `[b`.

```
<randomUInt32>≡
def randomUInt32(a=0, b=(2**32 - 1)):
    return util.randint(a, b)
```

`randomIpAddress` returns as a string a randomly generated IPv4 address in dotted quad notation.

```
<randomIpAddress>≡
def randomIpAddress():
    return ".".join([ str(random.randint(0, 255)) for i in xrange(4) ])
```

`randomMacAddress` returns as a string a randomly generated MAC address, as six groups of two hexadecimal digits separated by `sep`.

```
<randomMacAddress>≡
def randomMacAddress(sep='-'):
    return sep.join([ hex(random.randint(0, 255))[2:] for i in xrange(6) ])
```

```
<randomByteSeq>
<randomAlphaNumStr>
```

`randomByteSeq` generates a random sequence of bytes. If both `a` and `b` are specified, the result is of random length in the range of `[a,b]`; if only `a` is given the length will be `a` exactly.

```
<randomByteSeq>≡
def randomByteSeq(a, b=None):
    if b is not None:
        length = util.randint(a, b)
    else:
        length = a

    return "".join([chr(util.randint(0,255))
                    for i in xrange(length)])
```

`randomAlphaNumStr` is like `randomByteSeq`, but generates a string of alphanumeric characters only instead of a random sequence of bytes.

```

<randomAlphaNumStr>≡
def randomAlphaNumStr(a, b=None):
    if b is not None:
        length = util.randint(a, b)
    else:
        length = a

    return "".join([
        random.choice(string.letters + string.digits)
        for i in xrange(length)])

```

`md5Hidden` returns the byte seq of `pwd` after it is hidden using MD5.

Computing the `MD5-hidden(pwd)` requires read-access to the `sharedSecret` of the client-server pair, and `requestAuthenticator` of the RADIUS packet where this hidden password would be inserted. The algorithm is explained in the RFC 2865. Some implementations we have seen assum that `len(pwd) <= 16`.

```

<MD5-hidden>≡
def md5Hidden(pwd, sharedSecret, authenticator):
    n = 16 - len(pwd) % 16
    if n < 16:
        pwd += '\x00' * n

    hidden = ""
    k = 0
    last = authenticator
    # TBD functional prog style
    for i in xrange(len(pwd) / 16):
        hash = md5.new(sharedSecret + last).digest()
        for j in xrange(16):
            hidden += chr(ord(hash[j]) ^ ord(pwd[j]))
        k += 1
        last = hidden[-16:]
    return hidden

```

```

<MD5-exposed>≡
def md5Exposed(ewd, sharedSecret, authenticator):
    TBD()

```

```

<Generators for each attribute>≡
<generate-Attribute-Generator>

```

```
TBD{  }
⟨generate-Attribute-Generator⟩≡
```

As we read the `radius-defs.txt`, we generate the definitions of the 60+ more of the attribute generating functions. See `defs.nw`.

`User-Name` is permitted to be a name upto 253 bytes, but handling of 63-byte-long names is recommended in the RFC. Note also that the RFC does explicitly permits all 256 of the bytes; the name need not be made of only visible characters.

```
TBD{  The generation of User-Password and User-Name need to be coupled.
}
```

9.2 General Value Generators

These are some common general forms for generating values that follow a pattern. Each of these functions creates and returns a closure suitable for giving a sequence of values with the specified parameters (and can thus be given in place of a constant value in our library).

```
⟨General Value Generators⟩≡
  ⟨monotonically increasing⟩
  ⟨random byte seq of length⟩
```

Integer values increasing monotonically from a given first value.

```
⟨monotonically increasing⟩≡
def monotonicIncrInts(initial):
    def inner():
        value = initial - 1
        while True:
            value += 1
            yield value
        pass
    return inner().next
```

Sequences of random bytes, each of exactly the given length.

```
⟨random byte seq of length⟩≡
def randomByteSeqsOfLength(length):
    def inner():
        return randomByteSeq(length)
    return inner
```

Chapter 10

Logging and Monitoring

In our design, we will assume that there is no shortage of disk space for logging even though in a single test we may send 100K RADIUS packets. We also assume that security and confidentiality of the logs is a non-issue.

Mining the logs for interesting bits is a well-known problem. In our context, we solve the problem simply as follows. When a test RADIUS packet is generated, we also predict what the response from an RADIUS server should be. We give this to a watcher thread. When an unmatched reply arrives, this is logged into an alert log.

```
<logs.py>≡  
# logs.py generated from logs.nw  
  
<main>
```

10.1 Logging

Every packet sent and received is logged. The ports to watch are the standard one (destination 1812), and the source port or ports (in case of running out of identifiers).

The log can be a database or a collection of plain text files. We do not expect speed problems even with the most naive implementation of the log facility.

It is useful to have a log format that is used by one or more well-know log analyzers.

If possible, we should avoid writing fresh code for this module. May be we can fork out an existing program written in Python or not.

10.2 Monitoring

Every `Channel` has a `receiver` and a `monitor` thread.¹ The `receiver` thread is in-charge of receiving all packets from the RADIUS server. The `monitor` thread maintains a collection of packets sent out by the `sender` thread. The `monitor` thread can predict the responses to these.

We assume that we have access to the same user database that the RADIUS server has.

We expect to write fresh code for this module.

TBD{ Learn `twisetd` and see if it is useful. }

`<main>`≡

¹Note to us: May be we can merge these two into one.

Chapter 11

Utility Functions

We collect here all mundane utility methods. But, we make good use of Python, and its library.

`invoke` robustly calls closures passed in by users. If the given object is not in fact callable, it is returned (instead of the value returned when it was called) and no exceptions are raised.

```
<util.invoke>≡  
def invoke(func, *args):  
    if callable(func):  
        return func(*args)  
    else:  
        return func
```

```
<util.invoke Tests>≡  
assert(invoke(None) == None)  
assert(invoke("any constant") == "any constant")  
assert(invoke(lambda : 5) == 5)
```

`dict_merge` is like the `update` method provided by Python for dictionaries, but will not clobber existing entries.

If `error` is not `False` (the default), a `KeyError` will be raised for duplicated keys.

```
<dict_merge>≡
def dict_merge(dest, source, error=False):
    for k,v in source.items():
        if k not in dest.keys():
            dest[k] = v
        elif error:
            e = "Duplicated: " + k
            raise KeyError(e)
```

`thisHostIpAddress` is intended to return the current IPv4 address of this host as a dotted quad stored in a string.

Unfortunately, the "current IP address" is a somewhat murky concept from the perspective of Python. More precisely, this function returns the address associated with the system's hostname. Be aware of the problems arising, for example, from race conditions, with multiple network interfaces, when the hostname is named with 127.0.0.1, etc.

```
<thisHostIpAddress>≡
def thisHostIpAddress():
    return socket.gethostbyname(socket.gethostname())
```

```
<isHostReachable>≡
def isHostReachable(host):
    try:
        socket.gethostbyname(host)
        return True
    except:
        return False
```

`noop` is a do nothing function.

It is used to initialize `self.gen` and `self.isa` for a new Attribute.

```
<noop>≡
def noop():
    return 1
```

This is a generic Exception from which all the exception classes in this library derive (so users can catch any exception we raise).

```
<Exception Classes>≡
class PktGenException(Exception): pass
class PktGenWarning(Warning): pass

class RadiusException(PktGenException): pass
```

Python's `random.randint` requires that $a \leq b$. This wrapper avoids that constraint.

We use it for robustness when it is not obvious that Python's precondition will be met. This is especially useful, for example, when the length of a packet is one end of the range, as the length will not always be predictably greater or lesser than the other end.

```
<randint>≡
def randint(a, b):
    return random.randint(min(a,b), max(a, b))
```

Python's `random.sample` only returns a list of unique values.

```
<randomSample>≡
def randomSample(population, count, uniquify=False):
    if uniquify:
        return random.sample(population, count)
    else:
        return [ random.choice(population) for i in xrange(0, count) ]
```

Remainder of utility functions TBDocumented.

```

<util.py>≡
# util.py generated from util.nw

import struct
import socket

import random
random.seed()

<Exception Classes>

def TBD():
    return "to-be-done"

def string_shuffle(string):
    """random.shuffle() is destructive; its return value isn't the
    shuffled string. --Ben"""
    sl = list(string)
    random.shuffle(sl)
    return ''.join(sl)

def string_maybe_shuffle(string):
    """
    @return: Some permutation of the string, being a non-shuffled
             version at least 50% of the time. --Ben
    @rtype: string
    """
    return random.choice((string, string_shuffle(string)))

<isHostReachable>
<thisHostIpAddress>
<Address Utilities>

TIME1970 = 2208988800L # in seconds since 1900-01-01 00:00:00

def currentTime():
    """
    @return: The current date in seconds since the Unix TBD epoch.
    @rtype: int
    """
    return int(time.time())

def tightlyPack(args):

```

```
    # args is possibly heterogeneous list of items
    return tps

def packInteger(num):
    return struct.pack("!I", num)

def packDate(num):
    return struct.pack("!I", num)

def unpackInteger(num):
    return (struct.unpack("!I", num))[0]

def unpackDate(num):
    return (struct.unpack("!I", num))[0]

<util.invoke>
<dict_merge>
<randint>
<randomSample>
<noop>

<Tests and Demo>
# -eof-
```

<Address Utilities>≡

```
class AddressError(PktGenException): pass

def packIPAddress(ipAddress):
    try:
        return struct.pack("BBBB", *[ int(b)
                                     for b in ipAddress.split(".") ])
    except:
        raise AddressError("Bad IP format: %s." % ipAddress)
    pass

def unpackIPAddress(ipAddress):
    if len(ipAddress) != 4:
        raise AddressError("Bad IP format (%i bytes)." % len(ipAddress))
    return socket.inet_ntoa(ipAddress)

def packMACAddress(macAddress):
    bytes = macAddress.split(":")
    if len(bytes) != 6:
        raise AddressError("Bad MAC format: %s." % macAddress)
    return struct.pack("BBBBBB", *[ int(b, 16) for b in bytes ])

def unpackMACAddress(bytes):
    if len(bytes) != 6:
        raise AddressError("MAC byte sequence too short")
    bytes = [ hex(ord(b))[2:].upper() for b in bytes ]
    for i in xrange(len(bytes)):
        if len(bytes[i]) == 1:
            bytes[i] = "0" + bytes[i]
        pass
    pass
    return ":".join(bytes)
```

```
<Address Utility Tests>≡
assert(packIPAddress("0.14.09.155") == "\x00\x0E\x09\x9B")
try:
    packIPAddress("0.14.09.")
    assert(False)
except AddressError:
    pass

assert(unpackIPAddress("\x52\xF0\x14\xCC") == "82.240.20.204")
try:
    unpackIPAddress("\x00\x01\x02\x03\x04")
    assert(False)
except AddressError:
    pass

assert(packMACAddress("00:10:22:BB:10:A8") == "\x00\x10\x22\xBB\x10\xA8")
try:
    packMACAddress("00:")
    assert(False)
except AddressError:
    pass

assert(unpackMACAddress("\x0A\xA0\x22\xBB\x10\xA8") == "0A:A0:22:BB:10:A8")
try:
    unpackMACAddress("7654321")
    assert(False)
except AddressError:
    pass

<Tests and Demo>≡
if __name__ == "__main__":
    <util.invoke Tests>
    <Address Utility Tests>

    print "PASS"
    pass
```

Chapter 12

Conclusion

Chapter 13

References

References

BIONDI, P. 200x. Scapy. <http://www.secdev.org/projects/scapy/>.

INTERLINK NETWORKS. radpwtst: For testing authentication. <http://docs.hp.com/en/T1428-90056/index.html>.

MATETI, P. 2005. RADIUS protocol annotated in OM. Tech. rep., Wright State U. May.

<http://www.tummy.com/Community/software/radiuscontext/>

<http://www.zope.org/Members/Zen/ZRadius> Authenticator for Zope and Python. This product is now included as a component of exUserFolder. Can be used with the GenericUserFolder or LoginManager products to authenticate Zope users via a Radius server.

<http://py-radius.sourceforge.net/> radius.py is a pure Python module implementing client side RADIUS authentication.

<http://sourceforge.net/projects/pylibpcap/> Python module for the libpcap packet capture library, based on the original python libpcap module by Aaron Rhodes. Posted By: wiml Date: 2005-07-03 17:08 Summary: pylibpcap-0.5 released The first release of pylibpcap in three years! It's not a very fast-moving project. This release works with newer vesions of libpcap and SWIG, and should compile on a wider variety of operating systems and OS versions.

working draft/ October 10, 2006

Appendix A

GUI Script Creator

This interface generates code for a scripts using a subset of our library's features. It primarily focuses on generating valid or nearly valid packets.

This program requires version 2.4 or higher of the pygtk2 GTK+ bindings for Python from <http://www.pygtk.org>.

```
<ui.py>≡
#!/usr/bin/env python
# ui.py generated from ui.nw
import sys
import os

import pygtk
pygtk.require("2.0")
import gtk
import gtk.glade

from radiuspktgen.channel import Channel, RAND_ATTR_NAME, StandardRadiusDefs
from radiuspktgen.util import random

<ATTR_VAL enum class>
<class AttributeFormat>
<class RadiusPktGenGUI>

if __name__ == "__main__":
    win = RadiusPktGenGUI()
    gtk.main()
    pass
```

These are symbolic names of the different interpretations an `AttributeFormat` can have.

```
<ATTR_VAL enum class>≡
class ATTR_VAL:
    ARGV, RAND, INCR, SPECIFIC, DEFAULT = xrange(5)
    pass
```

An `AttributeFormat` object names an attribute and describes a form for its value. The form can be a constant value or one of following one of several patterns.

```
<class AttributeFormat>≡
class AttributeFormat:
    def __init__(self, name, interpret, param):
        self.name = name
        self.interpret = interpret
        self.param = param
        pass

    <Pretty printing>
    <Attribute creation code generation>
    pass
```

`AttributeFormat.gen_value` outputs code to generate a value of the form described by the `AttributeFormat`.

Here and elsewhere, we assume that all data ob an attribute formatter object was validated when it was read in from the dialog.

```
<Attribute creation code generation>≡
def gen_value(self):
    if self.interpret == ATTR_VAL.ARGV:
        return("sys.argv[%s]"
               % self.param)
    elif self.interpret == ATTR_VAL.INCR:
        return("attributeGen.monotonicIncrInts(%s)"
               % self.param)
    elif self.interpret == ATTR_VAL.RAND:
        return("attributeGen.randomByteSeqsOfLength(%s)"
               % self.param)
    elif self.interpret == ATTR_VAL.SPECIFIC:
        return("\'%s\'"
               % self.param)
    elif self.interpret == ATTR_VAL.DEFAULT:
        return ""
    pass
```

A textual explanation of each attribute formatter is displayed in the GUI.

```

<Pretty printing>≡
def __str__(self):
    if self.interpret == ATTR_VAL.ARGV:
        return("argv[%s]"
               % self.param)
    elif self.interpret == ATTR_VAL.INCR:
        return("Integers increasing monotonically from %s"
               % self.param)
    elif self.interpret == ATTR_VAL.RAND:
        return("%s random byte(s)"
               % self.param)
    elif self.interpret == ATTR_VAL.SPECIFIC:
        return("\\"%s\\"
               % self.param)
    elif self.interpret == ATTR_VAL.DEFAULT:
        return "Default"
    pass

```

The script creator interface has three tabs. The first tab contains parameters for a **Channel** (client/server connection object). The second tab allows definition of packets. Finally, in the third tab one or more tests to be run over the channel can be defined in terms of these packets. (Packets and attributes are actually generators, or “templates”, in that their values may be set to vary according to some common patterns). Finally, the “Save” button writes the equivalent code into a script using our library.

```

<class RadiusPktGenGUI>≡
class RadiusPktGenGUI:
    main_win_name = "main_window"
    attr_win_name = "attr_def_window"
    packets = {}
    channel_attrs = []
    argc_needed = 0
    run_indent = False

    <RadiusPktGenGUI.__init__>
    <Server tab callbacks>
    <Packets tab callbacks>
    <Tests tab>
    <Attribute Entry dialog>
    <Error message dialog>
    <Code generation and output>
    pass

```

Our GUI is mostly defined in a Glade XML file. Here we do additional set up for some of those widgets, and save references to ones we need to refer to later.

```

<RadiusPktGenGUI...init...>≡
def __init__(self):
    # Read the glade files and get the main windows
    self.main_widgets = gtk.glade.XML("ui.glade", self.main_win_name)
    self.main_win = self.main_widgets.get_widget(self.main_win_name)

    self.attr_widgets = gtk.glade.XML("ui.glade", self.attr_win_name)
    self.attr_win = self.attr_widgets.get_widget(self.attr_win_name)
    self.attr_win.set_transient_for(self.main_win)

    # FIXME: tab labels from glade?
    nb = self.main_widgets.get_widget("main_window_nb")
    nb.set_tab_label_text(self.main_widgets.get_widget("conn_tab_vbox"),
                          "Server Connection")
    nb.set_tab_label_text(self.main_widgets.get_widget("pkt_tab_vbox"),
                          "Packets")
    nb.set_tab_label_text(self.main_widgets.get_widget("tests_tab_vbox"),
                          "Tests")

    # Hook up signals
    signals = {}
    signals["on_main_window_destroy"] = gtk.main_quit
    signals["on_quit_clicked"] = gtk.main_quit
    signals["on_common_attr_clicked"] = self.common_attr_cb
    signals["on_save_clicked"] = self.write_script_cb
    signals["on_pkt_add_clicked"] = self.pkt_add_cb
    signals["on_pkt_edit_clicked"] = self.pkt_edit_cb
    signals["on_pkt_rm_clicked"] = self.pkt_rm_cb
    signals["on_test_add_clicked"] = self.test_add_cb
    signals["on_test_rm_clicked"] = self.test_rm_cb
    self.main_widgets.signal_autoconnect(signals)
    signals = {}
    signals["on_attr_confirm_clicked"] = self.attr_win_confirm_cb
    signals["on_add_attr_clicked"] = self.attr_add_cb
    signals["on_rm_attr_clicked"] = self.attr_rm_cb
    signals["on_attr_val_apply_clicked"] = self.attr_val_apply_cb
    signals["on_attr_used_changed"] = self.attr_used_select_cb
    signals["on_add_attr_rand_clicked"] = self.attr_add_rand_cb
    self.attr_widgets.signal_autoconnect(signals)

    # Set up the list of attributes available for adding
    self.avail_attrs = self.attr_widgets.get_widget("avail_attrs_combo")
    self.fill_avail_attrs()

```

```

# Set up the list of attributes currently added
self.used_attrs = self.attr_widgets.get_widget("attr_used_list")
self.used_attrs_store = gtk.ListStore(str, int, str, str)
self.used_attrs.set_model(self.used_attrs_store)
column = gtk.TreeViewColumn()
column.set_title("Attribute Name")
self.used_attrs.append_column(column)
cell = gtk.CellRendererText()
column.pack_start(cell)
column.add_attribute(cell, "text", 0)
# Interpretation column (see AttributeFormat.__str__())
column = gtk.TreeViewColumn()
column.set_title("Value")
self.used_attrs.append_column(column)
cell = gtk.CellRendererText()
column.pack_start(cell)
column.add_attribute(cell, "text", 3)

# We access these later
self.attr_val_argv_rb = \
    self.attr_widgets.get_widget("attr_val_argv_rb")
self.attr_val_incr_rb = self.attr_widgets.get_widget("attr_val_incr_rb")
self.attr_val_rand_rb = self.attr_widgets.get_widget("attr_val_rand_rb")
self.attr_val_specific_rb = \
    self.attr_widgets.get_widget("attr_val_specific_rb")
self.attr_val_default_rb = \
    self.attr_widgets.get_widget("attr_val_default_rb")
self.attr_value_entry = self.attr_widgets.get_widget("attr_value_entry")
self.get_timeout_int \
    = self.main_widgets.get_widget("timeout_spin").get_value_as_int
self.get_pkt_count_int \
    = self.main_widgets.get_widget("pkt_count_spin").get_value_as_int
self.get_delay_int = \
    self.main_widgets.get_widget("delay_spin").get_value_as_int
self.get_server_text = \
    self.main_widgets.get_widget("server_name_entry").get_text
self.get_secret_text = \
    self.main_widgets.get_widget("secret_entry").get_text
self.get_auth_port_text = \
    self.main_widgets.get_widget("auth_port_entry").get_text

# Set up the list of currently defined packets
self.pkt_combo = self.main_widgets.get_widget("pkt_combo")
self.pkt_store = gtk.ListStore(str)
self.pkt_combo.set_model(self.pkt_store)

```

```

cell = gtk.CellRendererText()
self.pkt_combo.pack_start(cell)
self.pkt_combo.add_attribute(cell, "text", 0)

# ...and show the same list as available packets in the Tests tab
self.test_pkt_combo = self.main_widgets.get_widget("test_pkt_combo")
# Share one packet store!
self.test_pkt_combo.set_model(self.pkt_store)
cell = gtk.CellRendererText()
self.test_pkt_combo.pack_start(cell)
self.test_pkt_combo.add_attribute(cell, "text", 0)

# Set up the rest of the Tests tab
self.test_combo = self.main_widgets.get_widget("test_combo")
self.test_store = gtk.ListStore(str, int, int)
self.test_combo.set_model(self.test_store)
for col in xrange(0, 3):
    cell = gtk.CellRendererText()
    self.test_combo.pack_start(cell)
    self.test_combo.add_attribute(cell, "text", col)
    pass

pass

```

The **Channel** must have a RADIUS server hostname or IP address, authentication port, and shared secret. Additionally, “common attributes” which will be added to every packet sent over the channel may be defined here.

```

<Server tab callbacks>≡
def common_attr_cb(self, *args):
    self.channel_attrs = \
        self.dialog_to_attributes(load=self.channel_attrs)
pass

```

At the top of the “Packets” tab is an (initially empty) list of defined packet generators. Below are controls for creating packet generators or editing the one currently selected in the list.

A packet generator is defined here as a sequence of zero or more AttributeFormat objects. Each packet is given a name for the purposes of identifying it for use in tests. This name will be used as an identifier in the Python code output, so it should not containing spaces, etc. TBD{ not yet checked by the GUI. }

A packet’s authenticator and identifier appear with its attributes. All the same generation methods can be used.

TBD{ The special attributes for authenticator and identifier can be deleted. To prevent that, they would need to be special-cased in `attr_rm_cb()`. They become default if not specified otherwise, so generation will still work. However, there is no way to get them back for a specific packet. }

<Packets tab callbacks>≡

```
def pkt_add_cb(self, *args):
    name = self.main_widgets.get_widget("pkt_name_entry").get_text()
    if name == "":
        return self.error_dialog("Cannot add packet without a name.")
    elif name in self.packets.keys():
        return self.error_dialog("Packet name must be unique.")
    elif name.find(" ") >= 0:
        return self.error_dialog("Invalid packet name.")

    load = [AttributeFormat("<Identifier>",
                            ATTR_VAL.DEFAULT,
                            ""),
            AttributeFormat("<Authenticator>",
                            ATTR_VAL.DEFAULT,
                            "")
            ]
    self.packets[name] = self.dialog_to_attributes(load=load)
    add = self.pkt_store.append([name])
    self.pkt_combo.set_active_iter(add)
    pass

def pkt_edit_cb(self, *args):
    name = self.pkt_combo.get_active_text()
    if name is None:
        return self.error_dialog("You must select a packet.")
    self.packets[name] = \
        self.dialog_to_attributes(load=self.packets[name])
    # FIXME: Cannot edit the name. Confusing?
    pass
```

```
def pkt_rm_cb(self, *args):
    name = self.pkt_combo.get_active_text()
    if name is None:
        return self.error_dialog("You must select a packet.")
    self.packets.pop(name)
    active = self.pkt_combo.get_active_iter()
    if active is not None:
        self.pkt_store.remove(active)
        pass
    pass
```

A test triplet, defined in the “Tests” tab, consists of a packet generation definition (referencing the packet tab), the number of packets to generate and send, and a delay in between. Previously defined tests are displayed in an (initially empty) list where they may be reviewed or deleted.

```

<Tests tab>≡
def test_add_cb(self, *args):
    packet_name = self.test_pkt_combo.get_active_text()
    if packet_name is None:
        return self.error_dialog("You must select a packet.")
    count = self.get_pkt_count_int()
    delay = self.get_delay_int()
    add = self.test_store.append([packet_name, count, delay])
    self.test_combo.set_active_iter(add)
    pass

def test_rm_cb(self, *args):
    active = self.test_combo.get_active_iter()
    if active is None:
        return self.error_dialog("You must select a test to remove.")
    else:
        self.test_store.remove(active)
        pass
    pass

def tests_to_list(self):
    tests = []
    def walker(model, path, iter, tests):
        test_form = [model.get_value(iter, 0),
                    model.get_value(iter, 1),
                    model.get_value(iter, 2)]
        tests.append(test_form)
        pass
    self.test_store.foreach(walker, tests)
    return tests

```

The attribute entry dialog allows definition of a sequence of `AttributeFormat` objects. `dialog_to_attributes` runs the dialog (optionally with default values specified as a list of `AttributeFormat` objects to load into the dialog), then returns a list of `AttributeFormat` objects representing the test writer’s input. We deliberately do not prevent attributes from being specified multiple times or in other invalid ways.

To specify value generation and have the attribute name/code randomly selected, use the “Add Random” button.

Currently, every change to an attribute value generator must be explicitly saved with the “Apply” button. TBD{ Make it automatic. }

<Attribute Entry dialog>≡
<Read Attribute Definitions>
<Attribute selection>
<Other Attribute Callbacks>

```
def attributes_to_dialog(self, attributes):
    self.used_attrs_store.clear()
    for attr_form in attributes:
        self.used_attrs_store.append([attr_form.name,
                                     attr_form.interpret,
                                     attr_form.param,
                                     attr_form])
    pass
pass

def dialog_to_attributes(self, load=[]):
    self.attributes_to_dialog(load)
    self.attr_win.run()
    attributes = []
    def walker(model, path, iter, attributes):
        attr_form = AttributeFormat(name=model.get_value(iter, 0),
                                    interpret=model.get_value(iter, 1),
                                    param=model.get_value(iter, 2))
        attributes.append(attr_form)
    pass
    self.used_attrs_store.foreach(walker, attributes)
    return attributes
```

We use the attribute definitions our library reads into `StandardRadiusDefs` (see chapter 4). The user can select an attribute name by typing with auto-completion or opening the drop-down list.

```
<Read Attribute Definitions>≡
def fill_avail_attrs(self):
    avail_attrs_store = gtk.ListStore(str)
    self.avail_attrs.set_model(avail_attrs_store)
    self.defined_attrs = StandardRadiusDefs.byname.keys()
    self.defined_attrs.sort()
    for name in self.defined_attrs:
        avail_attrs_store.append([name])
    pass

    completer = gtk.EntryCompletion()
    completer.set_model(avail_attrs_store)
    completer.set_minimum_key_length(1)
    completer.set_text_column(0)
    self.avail_attrs.child.set_completion(completer)
    self.avail_attrs.set_text_column(0)
    pass
```

```

<Other Attribute Callbacks>≡
def attr_win_confirm_cb(self, *args):
    self.attr_win.hide()
    pass

def attr_used_select_cb(self, *args):
    model, active = self.used_attrs.get_selection().get_selected()
    if active == None:
        return

    interpret = model.get_value(active, 1)
    if interpret == ATTR.VAL.ARGV:
        self.attr_val_argv_rb.set_active(True)
    elif interpret == ATTR.VAL.INCR:
        self.attr_val_incr_rb.set_active(True)
    elif interpret == ATTR.VAL.RAND:
        self.attr_val_rand_rb.set_active(True)
    elif interpret == ATTR.VAL.SPECIFIC:
        self.attr_val_specific_rb.set_active(True)
    elif interpret == ATTR.VAL.DEFAULT:
        self.attr_val_default_rb.set_active(True)
    pass

    param = model.get_value(active, 2)
    self.attr_value_entry.set_text(param)
    pass

def attr_val_apply_cb(self, *args):
    model, active = self.used_attrs.get_selection().get_selected()
    if active == None:
        return self.error_dialog("No attribute in use selected to edit.")

    interpret = None
    if self.attr_val_argv_rb.get_active():
        interpret = ATTR.VAL.ARGV
    elif self.attr_val_incr_rb.get_active():
        interpret = ATTR.VAL.INCR
    elif self.attr_val_rand_rb.get_active():
        interpret = ATTR.VAL.RAND
    elif self.attr_val_specific_rb.get_active():
        interpret = ATTR.VAL.SPECIFIC
    elif self.attr_val_default_rb.get_active():
        interpret = ATTR.VAL.DEFAULT
    pass

```

```
param = self.attr_value_entry.get_text()
if (interpret == ATTR_VAL.ARGV
    or interpret == ATTR_VAL.INCR
    or interpret == ATTR_VAL.RAND):
    try:
        param = int(param)
    except:
        return self.error_dialog("Parameter must be an integer.")
    pass

if interpret == ATTR_VAL.ARGV and self.argc_needed < param:
    self.argc_needed = param
    pass

# Attribute name (column 0) does not change
model.set_value(active, 1, interpret)
model.set_value(active, 2, param)
model.set_value(active, 3,
                AttributeFormat(name=model.get_value(active, 0),
                                interpret=interpret,
                                param=param))

pass
```

Attributed names are deliberately left in the available list so that an attribute can be included multiple times by the test writer.

<Attribute selection>≡

```
def attr_add(self, name):
    add = self.used_attrs_store.append([name, ATTR_VAL.SPECIFIC, "", "\\\""])
    self.used_attrs.get_selection().select_iter(add)
    pass

def attr_add_cb(self, *args):
    active = self.avail_attrs.get_active_text()
    if active not in self.defined_attrs:
        # FIXME: I find it tempting to select an attr in the combo
        # box and start editing but not add it. make that auto-add
        # instead of error?
        return self.error_dialog("Unknown attribute \"%s\"." % active,
                                parent=self.attr_win)
    self.attr_add(active)
    pass

def attr_add_rand_cb(self, *args):
    self.attr_add(RAND_ATTR_NAME)
    pass

def attr_rm_cb(self, *args):
    model, active = self.used_attrs.get_selection().get_selected()
    if active is not None:
        model.remove(active)
        pass
    pass
```

Convenience method to display error dialogs. Parent defaults to the main window.

```
<Error message dialog>≡
def error_dialog(self, message, parent=None):
    if parent is None:
        parent = self.main_win
        pass
    md = gtk.MessageDialog(parent=parent,
                           flags=0,
                           type=gtk.MESSAGE_ERROR,
                           buttons=gtk.BUTTONS_CLOSE,
                           message_format=message)

    md.run()
    md.destroy()
    pass
```

We now generate Python code which calls into our library to generate the user's packets and perform the tests.

```

<Code generation and output>≡
  <Attributes code generation>
  <Packet code generation>
  <Save dialog>

def write_script_cb(self, *args):
    server = self.get_server_text()
    secret = self.get_secret_text()
    port = self.get_auth_port_text()
    timeout = self.get_timeout_int()
    if server == "":
        return self.error_dialog("Missing server name.")
    if secret == "":
        return self.error_dialog("Missing shared secret.")
    try:
        port = int(port)
    except:
        return \
            self.error_dialog("Invalid authentication port number.")

    filename = self.get_save_filename(server + ".py")
    if filename is None:
        return
    try:
        script = file(filename, "w")
    except IOError:
        return self.error_dialog("Couldn't write to %s." % filename)
    os.chmod(filename, 0755)

    # FIXME: generated code is ugly: line breaks in suboptimal places
    script.write("#!/usr/bin/env python\n"
        "import sys\n"
        "if len(sys.argv) <= %i:\n"
        "    print \"Error: needed at least %i command line \"
        "argument(s).\"\n"
        "    sys.exit()\n"
        "    pass\n"
        "% (self.argc_needed, self.argc_needed))

    script.write("from radiuspktgen.channel import Channel\n"
        "from radiuspktgen import attributeGen\n"
        "from radiuspktgen import packetUtil\n"
        "achannel = Channel("

```

```

        "server=\"%s\", \n"
        "sharedSecret=\"%s\", \n"
        "timeout=%i \n"
        ") \n"
        % (server, secret, timeout))
script.write("achannel.authport = %i \n" % port)
# Add channel common attributes
script.write("achannel.defineActions(\n"
            "%s \n"
            ") \n"
            % self.attributes_to_actions_code(self.channel_attrs))

for name, attributes in self.packets.items():
    self.write_packet(script, name, attributes)
    pass

# FIXME? if no tests are defined, is the correct behavior to
# run nothing, or let the default test be generated and run?
# Maybe warning dialog(s)?
script.write("achannel.conductTest([\n"
            for (packet, reps, delay) in self.tests_to_list():
                script.write("(%s, %s, %s), \n" % (packet, reps, delay))
                pass
            script.write("]) \n")

script.close()
if self.run_indent:
    # FIXME: this doesn't work everywhere, and when it does it is
    # suboptimal because of placement of newlines above.
    elisp = "(progn "
        " (with-current-buffer (file-name-nondirectory \"%s\")"
        " (python-mode)"
        " (indent-region (point-min) (point-max)))"
        " (save-buffer 0)))" % filename
    proc = os.popen("emacs -q -batch "
        "--file \"%s\" "
        "--eval '%s'" % (filename, elisp))
    pass
pass

```

<Attributes code generation>≡

```
def attributes_to_actions_code(self, attributes):
    attributeNamesGen = ["\"attributeNamesGen\": ["]
    code = [{""]

    for attr_form in attributes:
        if attr_form.name == RAND_ATTR_NAME:
            # FIXME? could instead vary which attribute name is
            # added to each packet.
            attr_form.name = random.choice(StandardRadiusDefs.byname.keys())
            pass
        code.append("\"%s\": %s,"
                    % (attr_form.name, attr_form.gen_value()))
        attributeNamesGen.append("\"%s\"," % attr_form.name)
        pass

    attributeNamesGen.append("],")
    code.append("%s" % "\n".join(attributeNamesGen))
    code.append("}")
    return "\n".join(code)

def attributes_to_pairs_code(self, attributes):
    code = [{""]
    for attr_form in attributes:
        code.append("(" + "\"%s\"," % attr_form.name, attr_form.gen_value())
        pass
    code.append("]")
    return "\n".join(code)
```

Save uses the standard GTK+ dialog to query the user for a filename, showing .py scripts that already exist. The default filename is “<server>.py”.

The dialog supports offering the user the option of running an external indent tool on the generated code, but (due to lack of an appropriate tool) this is currently disabled.

```

<Save dialog>≡
def get_save_filename(self, default_filename):
    save_dialog = \
        gtk.FileChooserDialog(title="Save script",
                              parent=self.main_win,
                              action=gtk.FILE_CHOOSER_ACTION_SAVE,
                              buttons=(gtk.STOCK_CANCEL,
                                       gtk.RESPONSE_CANCEL,
                                       gtk.STOCK_SAVE,
                                       gtk.RESPONSE_ACCEPT),
                              backend=None)
    save_dialog.set_do_overwrite_confirmation(True)

    save_dialog.set_current_name(default_filename)
    pyFilter = gtk.FileFilter()
    pyFilter.set_name("Python source code")
    pyFilter.add_pattern("*.py")
    save_dialog.set_filter(pyFilter)

    indent_check = gtk.CheckButton(label="Run Emacs indent on output")
    indent_check.set_active(self.run_indent)
    #save_dialog.set_extra_widget(indent_check)

    response = save_dialog.run()
    save_dialog.hide()
    self.run_indent = indent_check.get_active()
    filename = None
    if response == gtk.RESPONSE_ACCEPT:
        filename = save_dialog.get_filename()
        pass
    return filename

```

```

<Packet code generation>≡
def write_packet(self, script, name, attributes):
    authGen = False
    idGen = False
    for attr_form in attributes:
        if attr_form.name == "<Authenticator>":
            authGen = attr_form
        elif attr_form.name == "<Identifier>":
            idGen = attr_form
        pass
    pass
    # Can't modify the list while iterating through it, but must remove
    # before turning attributes into code.
    if authGen:
        attributes.remove(authGen)
        pass
    if idGen:
        attributes.remove(idGen)
        pass

    script.write("%s = " % name)
    script.write("packetUtil.genChannelPackets(\"aqm\", achannel,\n")
    script.write(self.attributes_to_pairs_code(attributes) + "\n,")
    for gen in (idGen, authGen):
        if gen and gen.interpret != ATTR_VAL.DEFAULT:
            script.write("%s,\n" % gen.gen_value())
        else:
            script.write("None,\n")
        pass
    pass
    script.write(")\n")

pass

pass

```

Appendix B

Testing Many RADIUS packets in Parallel

A `RadiusPktGen Channel` (currently) does synchronous IO, with one thread for receiving and one for sending. In other words, it can send one packet while receiving another one, but not send two or more packets at once. This limitation can be worked around by starting several threads, each with a separate `Channel` object.

This test script starts 30 channels to the same server in parallel. (Each channel of course uses a different UDP source port.) The effect is that the server quickly gets a large number of packets from the same client NAS, but never two with the same source port and identifier.

```
<manyPacket.py>≡
#!/usr/bin/env python
import sys
import threading

if (len(sys.argv) < 6):
    print("Usage: %s user passwd radius-server nas-port-number "
          "secret [ppphint] [nasname]" % sys.argv[0])
    sys.exit()

from radiuspktgen.channel import Channel
from radiuspktgen.packet import ChannelPacket

<genChannel>

withThread = True
if withThread:
```


The simultaneous sending test simply uses Python's built-in threading facilities to start many threads, each running the `manyTest()` function. In each thread, `manyTest()` runs a simple test, `(genPacket, 256, 0)`, which sends packet generated by 256 successive invocations of `genPacket()`.

Note that Freeradius has a default limitation of 32 threads for processing incoming packets. Once the entire server thread pool is busy, Freeradius will start dropping packets, and the test script slows down considerably as it waits for timeouts.

```

<Run simultaneous sending test>≡
def manyTest():
    achannel, genPacket = genChannel()
    achannel.conductTest([(genPacket, 256, 0)])
    pass

for i in xrange(0, 30):
    threading.Thread(target=manyTest).start()
    print "Thread ", i, " started."
    pass

for thread in threading.enumerate()[1:]:
    if thread is not threading.currentThread():
        thread.join()
        pass
    pass

```

There are actually two tests in this script. If `withThread` is `True`, the script runs the simultaneous sending test as described above. Changing `withThread` to `false` would do almost the same thing (`genChannel()` and `genPacket()` are the same), but instead of spawning threads with `manyTest()` it simply tests 8,000 of the packets in sequence.

```

<Run 8,000 packets from single source test>≡
    achannel.conductTest([(genPacket, 8000, 0)])

```

(A generic explanation of a single threaded radtest-like client is in §1.2.3.)

```

# LocalWords: ChannelPacket genChannel packetNum argv

```

Appendix C

Testing FreeRADIUS authenticator handling

FreeRADIUS caches requests and replies. If an incoming request has the same client, identifier, and authenticator as a cached request that FreeRADIUS has already processed, the server simply sends the cached reply instead of rechecking the new request. We will trick FreeRADIUS using two successive packets that are identical under this comparison.

The first packet, “goodPacket”, has the correct User-Name and User-Password for a user who the RADIUS server has been told to authorize.

```
<Generate good packet>≡
    goodPacket = ChannelPacket("aqm", achannel,
                               [("User-Name", "steve"),
                                ("User-Password", "testing")
                               ])

```

The “badPacket” request, on the other hand, has incorrect credentials and *should* result in an Access-Reject. However, we give it the same identifier and authenticator. If badPacket arrives after goodPacket with the same source IP and port, the cache may cause this packet to receive an Access-Accept.

```
<Generate bad packet>≡
    badPacket = ChannelPacket("aqm", achannel,
                               [("User-Name", "not steve"),
                                ("User-Password", "not testing")
                               ])
    badPacket.authenticator = goodPacket.authenticator
    badPacket.identifier = goodPacket.identifier

```


APPENDIX C. TESTING FREERADIUS AUTHENTICATOR HANDLING130

GNU RADIUS (which, like FreerADIUS, is derived from Livingston RADIUS) is also be fooled by this script. The independently-implemented OpenRADIUS is not.

Example run:

Packet to be sent Thu Apr 6 22:48:09 2006:

Radius aqm id=159

Authenticator=\x88\xd5\x80\xcd\x7f\xda\x14\x9e\x8f\x0b\xa3N\xcdV\x5G

Shared Secret=testing123

attr User-Name: len=7, value=steve

attr User-Password: len=9, value=testing

Reply received Thu Apr 6 22:48:09 2006 from 192.168.17.214:1812:

Radius aam id=159

Authenticator=\x03\x06-0\xe6\t\xfer\xefo\xb0\x1ff\x01X\xb1

Shared Secret=testing123

attr Framed-IP-Address: len=6 (really 6), value=127.0.0.9

Packet to be sent Thu Apr 6 22:48:09 2006:

Radius aqm id=159

Authenticator=\x88\xd5\x80\xcd\x7f\xda\x14\x9e\x8f\x0b\xa3N\xcdV\x5G

Shared Secret=testing123

attr User-Name: len=11, value=not steve

attr User-Password: len=13, value=not testing

Reply received Thu Apr 6 22:48:09 2006 from 192.168.17.214:1812:

Radius aam id=159

Authenticator=\x03\x06-0\xe6\t\xfer\xefo\xb0\x1ff\x01X\xb1

Shared Secret=testing123

attr Framed-IP-Address: len=6 (really 6), value=127.0.0.9