MARROW MAILER

Drumil Kaushik Bakhai (dkb300)

Saniya Sudhir Alekar (ssa428)

TABLE OF CONTENTS

Introduction

What is marrow mailer?

A highly efficient and modular mail delivery framework for Python 2.6+ and 3.2+, formerly called TurboMail.

Marrow Mailer is a Python library for sending emails from your application.

By using Marrow Mailer you can:

- Easily construct plain text and HTML emails.
- Use various types of mail delivery management strategies (immediate, deferred, or even multi-server).
- Use various transport methods like SMTP, Amazon SES, sendmail, or even via direct on-disk mbox/maildir for email delivery.
- Multiple simultaneous configurations for more targeted delivery.

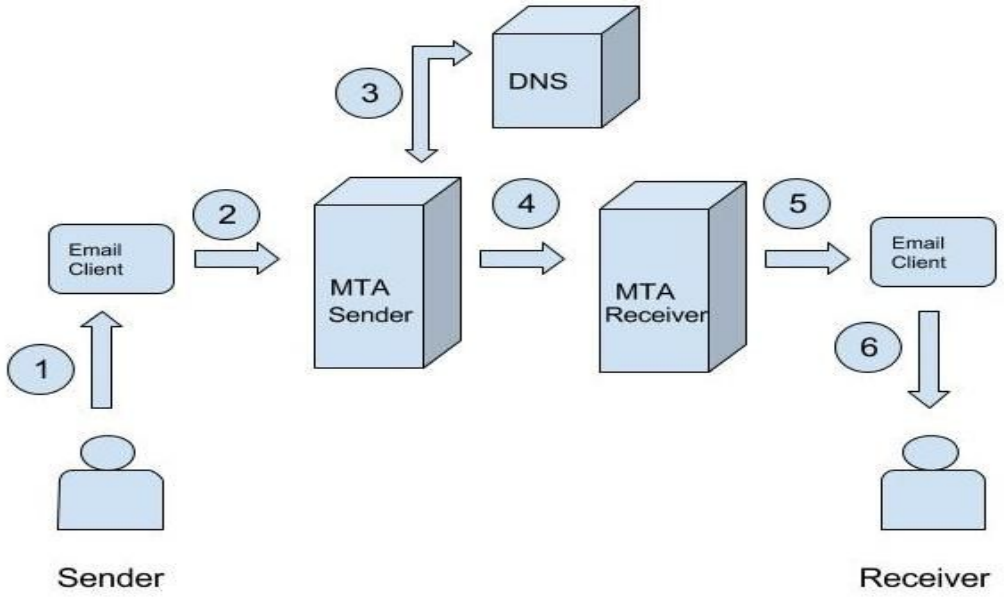
Marrow mailer utilizes the built in MIME message generation classes and SMTP. It also uses sendmail and various other ‘transport methods’.

For installation of marrow mailer, all one needs to do is type in ‘pip install marrow.mailer’ in the terminal or command prompt.

How does it work?

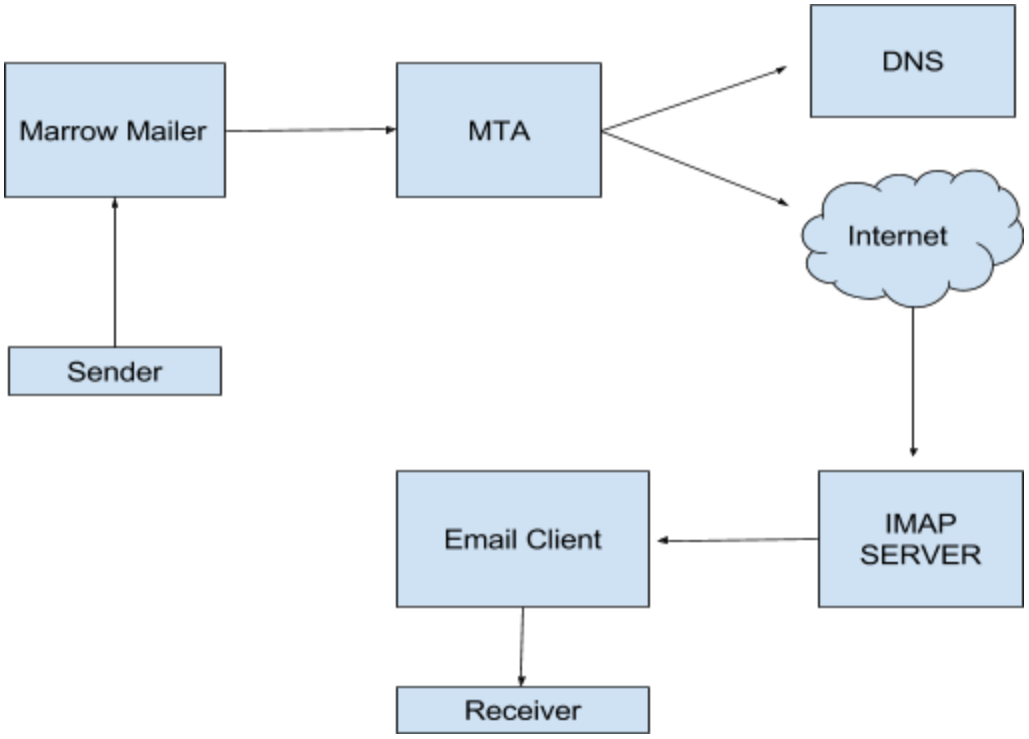
Sending an email:

When you compose an email and click the send button, your email client (Outlook, Gmail, etc) reads all the information: sender email address, receiver email address, subject line, email content, attachments and converts this into email text format.



Marrow Mailer:

Mailer is NOT an MTA like sendmail, postfix, qmail, etc. It is designed to deliver messages to a real mail server ("smart host") or other back-end which then actually delivers the messages to the recipient's server.



Basic Usage

```
from marrow.mailer import Mailer, Message

mailer = Mailer(dict(
        transport = dict(
                use = 'smtp',
                host = 'localhost')))
mailer.start()

message = Message(author="user@example.com", to="user-two@example.com")
message.subject = "Testing Marrow Mailer"
message.plain = "This is a test."
mailer.send(message)

mailer.stop()
```

As you can see in the above example, to use Marrow Mailer, you should:

- Instantiate a marrow.mailer.Mailer object with the configuration.
- Pass message instances to the Mailer instance's send() method.
- Configure multiple delivery mechanisms and choose, within your code, how you want each message delivered.

Mailer Methods

Method	Description
`__init__(config, prefix=None)`	Create and configure a new Mailer.
`start()`	Start the mailer. Returns the Mailer instance and can thus be chained with construction.
`stop()`	Stop the mailer. This cascades through to the active manager and transports.
`send(message)`	Deliver the given Message instance.
`new(author=None, to=None, subject=None, **kw)`	Create a new bound instance of Message using configured default values.

Message Methods

Method	Description
`__init__(author=None, to=None, subject=None, **kw)`	Create and populate a new Message. Any attribute may be set by name.
`__str__`	You can easily get the MIME encoded version of the message using the `str()` built-in.
`attach(name, data=None, maintype=None, subtype=None, inline=False)`	Attach a file (data=None) or string-like. For on-disk files, mimetype will be guessed.
`embed(name, data=None)`	Embed an image from disk or string-like. Only embed images!
`send()`	If the Message instance is bound to a Mailer instance, e.g. having been created by the `Mailer.new()` factory method, deliver the message via that instance.

Message Transports

Disk Transports

1. UNIX Mailbox
2. UNIX Mail Directory

Network Transports

1. Simple Mail Transfer Protocol (SMTP)
2. Internet Mail Access Protocol (IMAP)

Meta-Transports

1. Google AppEngine
2. Python Logging
3. Mock (Testing) Transport
4. Sendmail Command
5. Amazon Simple Email Service (SES)
6. SendGrid

Dependencies

1. Python DNS module
2. Pip

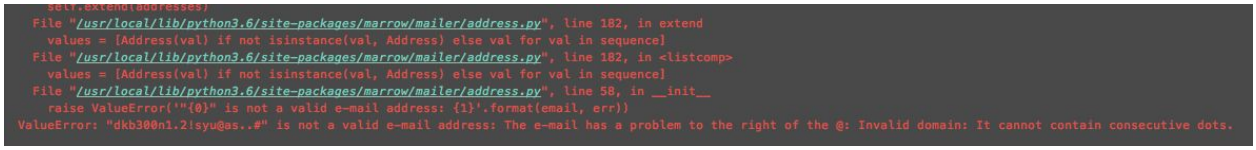
Threat model and approach

In this module we have tried to identify the possible ways to dismantle or attack the open source library. We consider the threats to occur from the User Input, Network Communication, Packages and libraries used inside the open source code.

User Input

The library asks the user's program to set up methods used to transport the message via internet. The user needs to enter the valid SMTP details when transport is selected. Additionally proper username and password for the gmail account through which SMTP is used needs to set up. After the credentials are verified, Mailer class checks for the correct format of email address, subjects, etc. As an attempt to validate format of email address we tried various combination of the same. Below is the screenshot of the code and error message. Additionally the marrow-mailer has included multiple unit test that is used to check for the same.

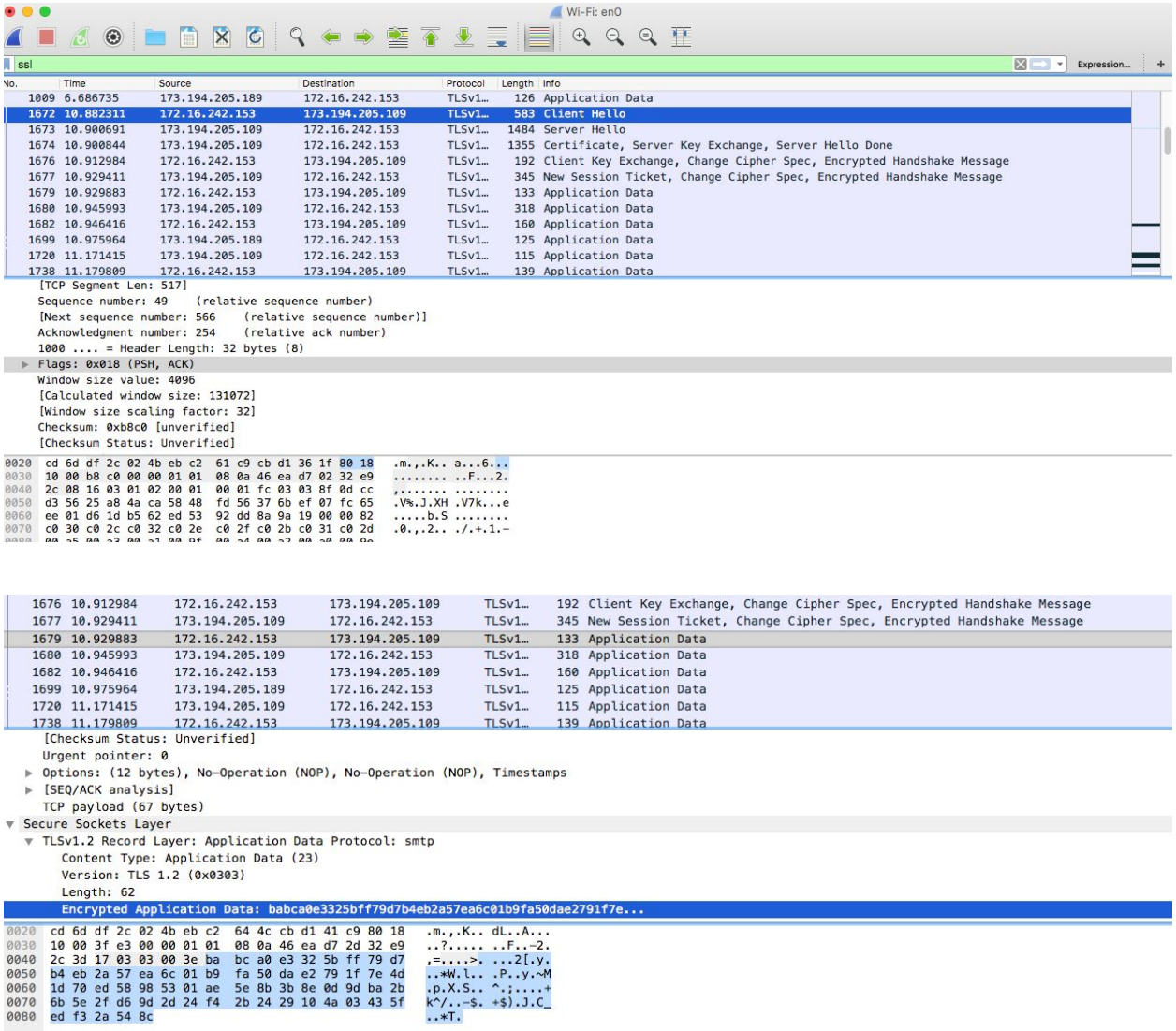
```
                    port='587')))
mailer.start()

message = Message(author="drums.b123@gmail.com", to="dkb300n1.2!syu@as..#")
message.subject = "?'\"asd@.CO$$~==√ôpœπ"
message.plain = "This is a test <html>" \
                "<body>" \
                "<h1> HELLOOOOO </h1>" \
                "</body></html>."
# message.attach("export-issue.py", data=None, maintype=None)
mailer.send(message)

mailer.stop()
```



```
    self.extend(addresses)
  File "/usr/local/lib/python3.6/site-packages/marrow/mailer/address.py", line 182, in extend
    values = [Address(val) if not isinstance(val, Address) else val for val in sequence]
  File "/usr/local/lib/python3.6/site-packages/marrow/mailer/address.py", line 182, in <listcomp>
    values = [Address(val) if not isinstance(val, Address) else val for val in sequence]
  File "/usr/local/lib/python3.6/site-packages/marrow/mailer/address.py", line 58, in __init__
    raise ValueError('"{0}" is not a valid e-mail address: {1}'.format(email, err))
ValueError: "dkb300n1.2!syu@as..#" is not a valid e-mail address: The e-mail has a problem to the right of the @: Invalid domain: It cannot contain consecutive dots.
```

Network Sniffing

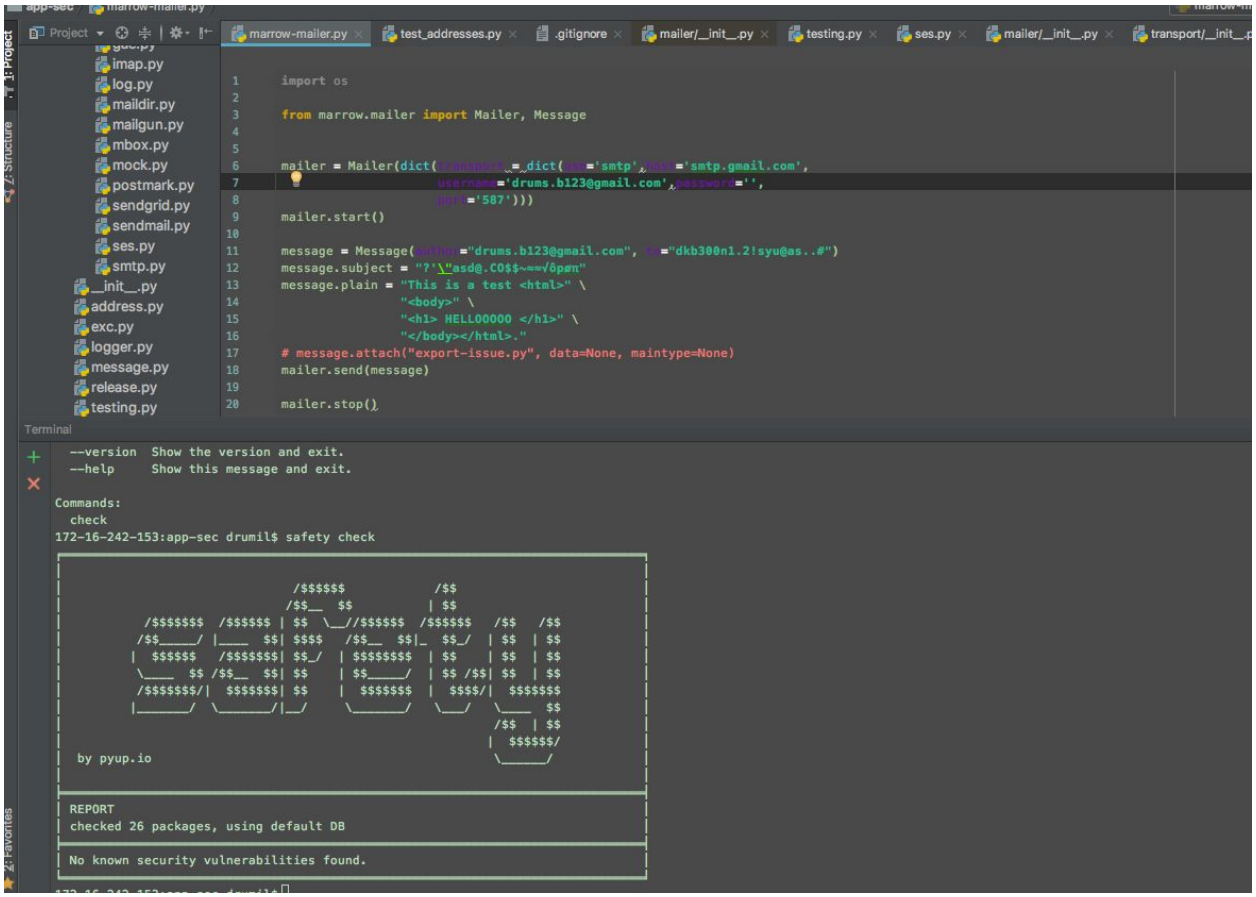
Since the user needs to authenticate SMTP server before they can start sending emails, it is necessary that all the sensitive information is securely transmitted over the internet. Therefore in order to perform man-in-the-middle attack the attacker can definitely sniff out the network traffic and correctly guess sensitive information. The marrow-mailer uses the TLS, SSL encryption to encrypt username and password. By analyzing the packets using Wireshark we can see that the data is encrypted when transferred over internet



From the above image we can see, after the Client Key Exchange is successful the user is sending the data to SMTP server (google). The data is been transferred in encrypted form and thereby reduces the chance of performing man-in-the-middle attack. However there are other vulnerabilities which are discussed in next section.

Dependencies

Marrow mailer uses multiple dependencies and libraries for its functionalities. The attacker can expose vulnerabilities in those dependencies to attack marrow-mailer. It is essential to lookup for insecure or malicious packages to make sure marrow-mailer is safe. Safety is a simple CL tool which checks for insecure packages used in the program. SafetyDB is list of packages name and all insecure releases as a plain list. The following screenshot confirms that marrow-mailer is not using any malicious or insecure packages.



Code auditing

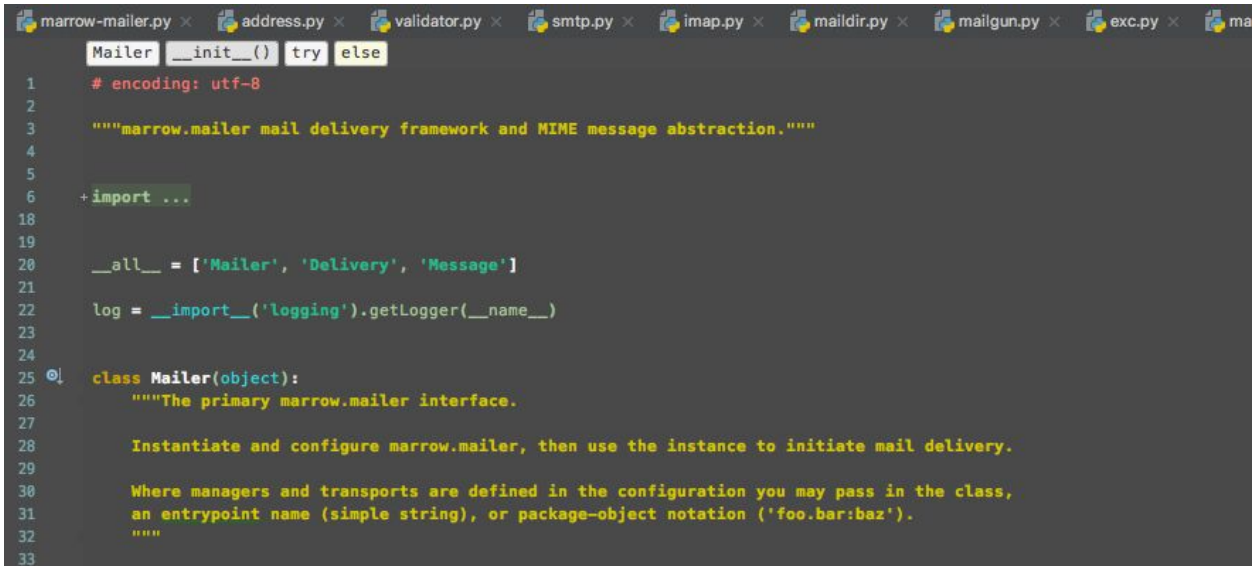
Based on the code audit, we discovered that flow of execution is easy to understand. The marrow-mailer library is:

- Modular
- Readable
- Well commented
- Structured test-driven development

We used PyCharm Code Editor and following steps were performed to understand code audit.

1. Reading documentation on GitHub and writing a sample program to check the functionalities.

2. Identifying main class which is called when the Mailer Instance is created. Init File usually contains the main class file and it acts as an entry point to the code.



3. Once the Mailer class is identified, we looked for various parameter the constructor or its method takes. Here from the documentation we passed an instance of the class Transport. The Transport class determines user's configuration whether to use SMTP or SendMail or any other mode of transport of messages.

4. The entire directory is dedicated to Transport. There are multiple files in transport. For manual review we have used the SMTP class. The smtp.py reads various configuration such as username, password, hostname, etc from the user. Afterwards, the class uses the native SMTP library to create SMTP connection and transfers the message.

```
SMTPTransport  __init__()
# encoding: utf-8

"""Deliver messages using (E)SMTP."""

import socket

from smtplib import (SMTP, SMTP_SSL, SMTPException, SMTPRecipientsRefused,
                     SMTPSenderRefused, SMTPServerDisconnected)

from marrow.util.convert import boolean
from marrow.util.compat import native

from marrow.mailer.exc import (
    TransportExhaustedException, TransportException, TransportFailedException,
    MessageFailedException)

log = __import__('logging').getLogger(__name__)


class SMTPTransport(object):
    """An (E)SMTP pipelining transport."""
```

5. Finally the user has to generate an entire message to send as an email. To, From, Subject, Body, cc, bcc are few characteristics we need in order to send an email. Marrow-mailer provides classes such as validator.py and message.py to validate user input.

```
Address  __init__()  if email is None  else
# encoding: utf-8

"""TurboMail utility functions and support classes."""

+import ...

__all__ = ['Address', 'AddressList']


class Address(object):
    """Validated electronic mail address class.

    This class knows how to validate and format e-mail addresses.  It uses
    Python's built-in `parseaddr` and `formataddr` helper functions and helps
    guarantee a uniform base for all e-mail address operations.

    The AddressList unit tests provide comprehensive testing of this class as
    well."""
```

6. Once user has created message and decided the mode of Transport, message can be send using “send” method of Mailer class.

```
def send(self, message):
    if not self.running:
        raise MailerNotRunning("Mail service not running.")

    log.info("Attempting delivery of message %s.", message.id)

    try:
        result = self.manager.deliver(message)

    except:
        log.error("Delivery of message %s failed.", message.id)
        raise

    log.debug("Message %s delivered.", message.id)
    return result
```

More threats that were discovered during code audit are mentioned in section below.

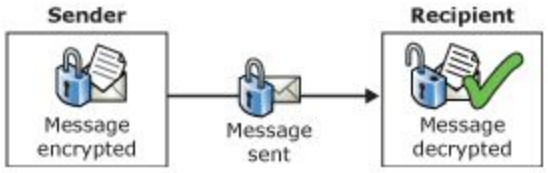
Vulnerabilities

ROBOT

Return Of Bleichenbacher's Oracle Threat (ROBOT) a 19 year old bug was discovered by security researchers which allowed the part of TLS private key to be displayed in the error message. In the marrow-mailer program there are options to use TLS and SSL encryption for sending the data over the network in a secured manner.

The flaw in RSA PKCS #1 v1.5 encryption affected almost top web domains. The error message generated during the process of padding for private key allowed an adaptive-chosen ciphertext. Therefore it completely breaks the confidentiality of TLS when it is used with RSA encryption. As we can see that error affected many new web servers and security researchers are trying to assess the risk of severe threat possess by faulty display of error messages. SSL and TLS are ubiquitously used for the generation of encrypted data and this small bug can generate immense amount of security countermeasure.

Using S/MIME (Secure Multipurpose Internet Mail Extensions) to secure emails



S/MIME renders anti-virus scanner useless. It's difficult to scan S/MIME incoming messages. Without client side malware detection for email, S/MIME becomes a problem.
If private keys are stored on the gateway server so that decryption occurs before the malware scan, unencrypted messages are then delivered to end users. Any message that an S/MIME email client stores encrypted cannot be decrypted if the applicable private key is unavailable or otherwise unusable. However, an untrusted certificate can still be used by hackers for cryptographic purposes. Malware will also get encrypted. Hence if mail is not scanned for malware anywhere but at the end points, such as a company's gateway, encryption will defeat the detector and successfully deliver the malware.
No security method used can be perfect. It is possible for hackers to impersonate a sender by obtaining the private information that is used for digital signatures. However, the S/MIME standard can handle these situations so that unauthorized signatures are shown to be invalid. Then again, all these security issues are not specific to S/MIME but most security mechanism that include digital signatures and message encryption.

Marrow Mailer has used the Python MIME modules and the functionalities it provides:

Importing MIME modules:

```
from datetime import datetime
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
from email.mime.nonmultipart import MIMENonMultipart
from email.utils import make_msgid, formatdate
from mimetypes import guess_type
```

MIME encoding:

```
def __str__(self):
        return self.mime.as_string()

__unicode__ = __str__

def __bytes__(self):
        return self.mime.as_string().encode('ascii')
```

MIME document (MIME encoded message):

```
def _mime_document(self, plain, rich=None):
        if not rich:
                message = plain

        else:
                message = MIMEMultipart('alternative')
                message.attach(plain)

                if not self.embedded:
                        message.attach(rich)

                else:
                        embedded = MIMEMultipart('related')
                        embedded.attach(rich)
                        for attachment in self.embedded:
                                embedded.attach(attachment)
                        message.attach(embedded)

        if self.attachments:
                attachments = MIMEMultipart()
                attachments.attach(message)
                for attachment in self.attachments:
                        attachments.attach(attachment)
                message = attachments

        return message
def mime(self):
        """Produce the final MIME message."""
        author = self.author
        sender = self.sender

        if not author:
                raise ValueError("You must specify an author.")

        if not self.subject:
                raise ValueError("You must specify a subject.")

        if len(self.recipients) == 0:
                raise ValueError("You must specify at least one recipient.")

        if not self.plain:
                raise ValueError("You must provide plain text content.")
```

Even though message encryption provides confidentiality, it does not authenticate the message sender in any way. An unsigned, encrypted message is as susceptible to sender impersonation as an unencrypted message. An encrypted message only shows that the message has not been altered from the original, though encrypted messages are said to provide data integrity.

SendMail vulnerabilities

Marrow mailer uses various “means of transport”. One of them is sendmail. The module uses sendmail and does not implement software version check.

The prescan() function:

The prescan() function in Sendmail 8.12.9 allows remote attackers to execute arbitrary code via buffer overflow attacks.
The prescan() function in the address parser in Sendmail does not accurately handle certain conversions from char and int types, which can cause a length check to be disabled, allowing attackers to cause a denial of service and possibly execute arbitrary code via a buffer overflow attack using messages.

Open mail relay:

“An **open mail relay** is an SMTP server configured in such a way that it allows anyone on the Internet to send email through it, not just mail destined to or originating from known users.”
Successful exploitation will allow attackers to send email messages outside of the served network. The flaw is due to an error in the mailconf module in Linuxconf which generates the Sendmail configuration file (sendmail.cf) and configures Sendmail to run as an open mail relay, which allows remote attackers to send spam email.

Other problems in the module

Incorrect documentation and bug in configuration parser

Problem:
The documentation for Marrow Mailer definitely needs some more work. The sample piece of code provided in ReadMe does not work unless modified. After running the code provided in ReadMe, an exception is thrown. This is also an open issue.

Ways to handle:
Sincere and detailed documentation and more unit testing would resolve the documentation problem. However by modifying the configuration parser so that it would be equipped to handle all types of configurations and not just the standard mail configurations. This can be does by including templates and then comparing the configuration to the stored templates.

Does not parse a comma in email address accurately

Problem:

While writing the code to send an email we came across a bug in the module code. This bug allows users to type a comma in the email address. However the module does not parse this well and this forces the address parser to parse the email address as two separate addresses.

For example, by typing in “John, Doe” john.doe@abc.com, the module considers John and Doe as two separate email addresses and tried to parse them and returns the following error:

ValueError: "John" is not a valid e-mail address: An email address must contain a single @

Ways to handle:

Modify the parsing algorithm to identify invalid and valid email addresses accurately. Splitting of strings using dots or commas or any other separator, would be useful. Using Python-NLTK (Natural Language Toolkit) would be useful for sentence parsing.

No validation of DNS lookup of email addresses in ‘validator.py’

```
import DNS

lookup_record = lookup_record.lower() if lookup_record else self._lookup_dns

if lookup_record not in ('a', 'mx'):
    raise RuntimeError("Not a valid lookup_record value: " + lookup_record)

if lookup_record == "a":
    request = DNS.Request(domain, **kw)

    try:
        answers = request.req().answers

    except (DNS.Lib.PackError, UnicodeError):
        # A part of the domain name is longer than 63.
        return False

    if not answers:
        return False

    result = answers[0]['data'] # This is an IP address

    if result in self.false_positive_ips: # pragma: no cover
        return False

    return result

try:
    return DNS.mxlookup(domain)

except UnicodeError:
    pass

return False
```

```
def __init__(self, fix=False, lookup_dns=None):
    self.fix = fix

    if lookup_dns:
        try:
            import DNS
        except ImportError: # pragma: no cover
            raise ImportError("To enable DNS lookup of domains install the PyDNS package.")

        lookup_dns = lookup_dns.lower()
        if lookup_dns not in ('a', 'mx'):
            raise RuntimeError("Not a valid *lookup_dns* value: " + lookup_dns)

    self._lookup_dns = lookup_dns
```

As seen in the code above, the module expects us to have the DNS python package installed. Otherwise the code will not perform DNS lookup. This leads to a success message even though the actual email address given does not exist.

However, the DNS package no longer exists in python; instead there is a PyDNS package which the module is prompting us to install.

No notification to user regarding message delivery or failure

We tried to use the module by sending an email to an invalid email address. While the email client (Outlook/Google, etc.) does inform us about whether the email address is valid or not, the python module does not inform us about message delivery success or failure. If an invalid email address is typed in the code, the module still returns no error, which in turn misleads the user and makes the user think that the message has been delivered successfully.

Message attachments through SendGrid has not been implemented yet

Email attachments can be sent through marrow mailer. However functionality to send email attachments through SendGrid has not been implemented yet.

```
if message.attachments:
    # Not implemented yet
    """
    attachments = []

    for attachment in message.attachments:
        attachments.append((
                attachment['Content-Disposition'].partition(';')[2],
                attachment.get_payload(True)
            ))

    msg.attachments = attachments
    """
    raise MailConfigurationException()
```

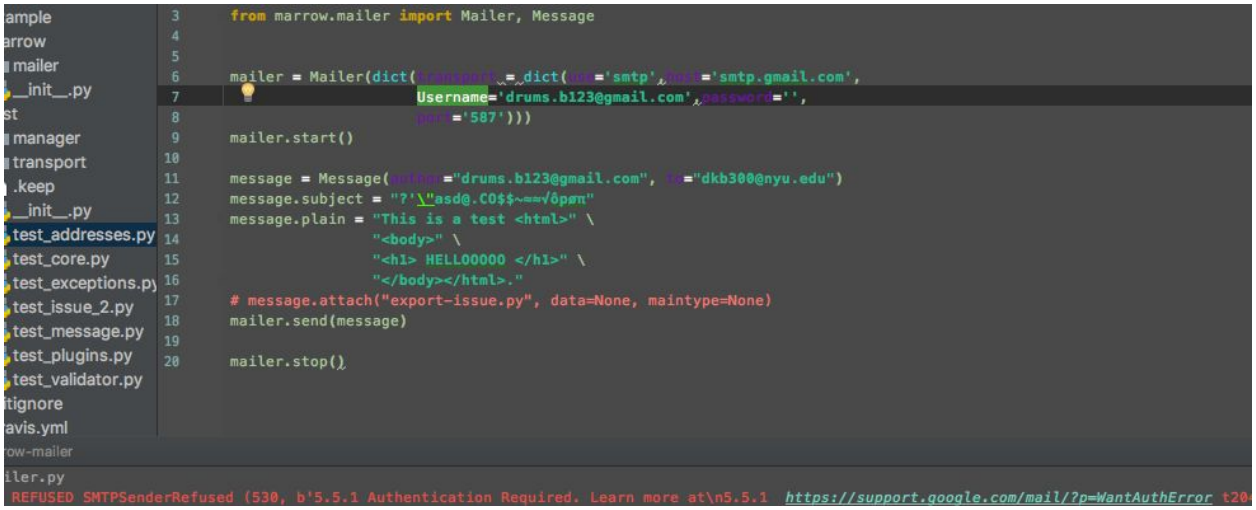
Boto Amazon SES

The Transport module provides users to setup Amazon web services Simple Email System SES to send data using Amazon's server. After carefully analyzing ses.py file we can see that it imports an older version of Boto. After having a quick lookup on Boto's GitHub code we can see that there are open version with respect to connection in Boto library. Vulnerability can be exploited using this open issue and instead of using the old version developers should use newer version Boto3 which is actively developed and maintained.

SMTP Configuration

The user needs to input username, password etc exactly like it is given in the documentation. However one cannot trust user input to this part. In the code if a developer passes configuration as **Username = “ ”** instead of **username = “ ”**, the error message saying SMTP authentication failed is displayed. Here even if the right credential are passed it will still display as SMTP authentication failed.

Therefore it is recommended to usually sanitize the tuple that is used for reading configuration parameter before passing to SMTP native library.




```
class SMTPTransport(object):
    """An (E)SMTP pipelining transport."""

    __slots__ = ('ephemeral', 'host', 'tls', 'certfile', 'keyfile', 'port', 'local_hostname', 'username', 'password',

    def __init__(self, config):
        self.host = native(config.get('host', '127.0.0.1'))
        self.tls = config.get('tls', 'optional')
        self.certfile = config.get('certfile', None)
        self.keyfile = config.get('keyfile', None)
        self.port = int(config.get('port', 465 if self.tls == 'ssl' else 25))
        self.local_hostname = native(config.get('local_hostname', '')) or None
        self.username = native(config.get('username', '')) or None
        self.password = native(config.get('password', '')) or None
        self.timeout = config.get('timeout', None)

        if self.timeout:
            self.timeout = int(self.timeout)

        self.debug = boolean(config.get('debug', False))

        self.pipeline = config.get('pipeline', None)
```

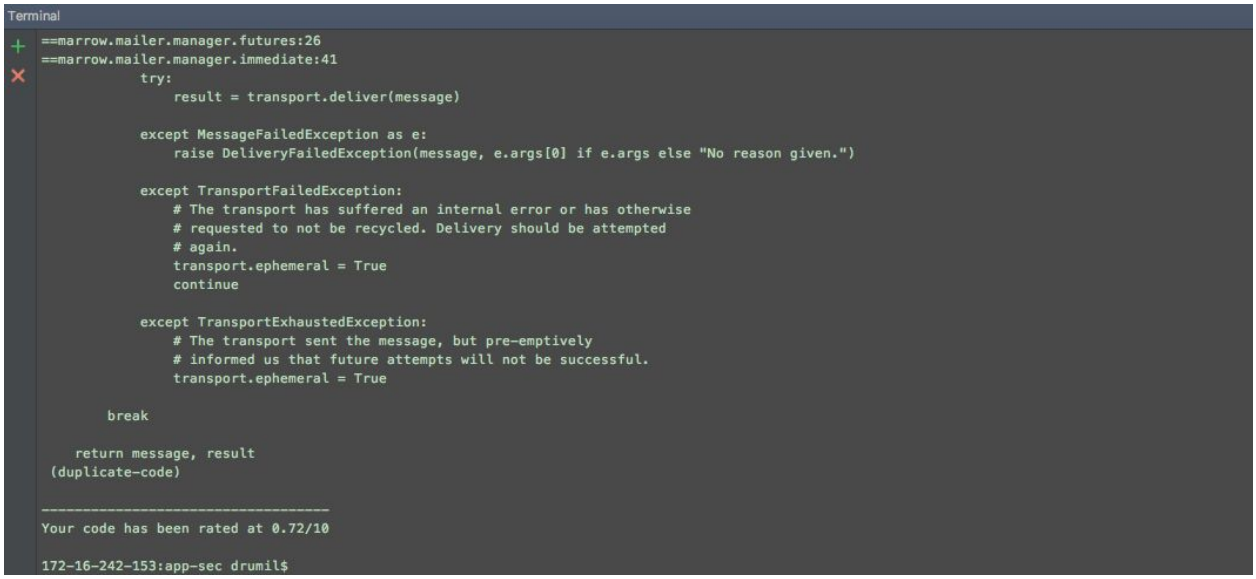
For security reason the password field is kept empty.

Best Practices

PyLint

We performed automated static analysis of code using PyLint which decides the quality of code with respect to standard coding practices in python.

1. Manually we installed pylint package using pip install pylint.
2. We passed the main directory as an argument to pylint command and following ratings were received.



Unit testing

The developers of marrow-mailers used test-driven development approach which is an ideal way to make sure that code is secured and bug-free. The Test driven development is a process to write test cases for the functionality before writing the logic of the same. This way the developer will handle cases for possible bugs in the functionality thereby saving time for testing during final release.

The developers have commented certain cases for unit testing such as the following.

The code is open to attackers and commenting out unit testing will give the attacker initial hint to attack the application.

Apart from few such cases the marrow-mailer has extensive unit testing and thereby it is less vulnerable to threats from bugs and errors

Conclusion

We have identified a lot of issues, few vulnerabilities and addressed many other problems related to Marrow Mailer. The module is otherwise a secure way of delivering messages in Python using various MTAs like smtplib, sendmail, etc.
The assessment and test made us realize that security mechanisms should be an important part of any project. Every project should be implemented with continuous integration and unit testing. Most of the issues we identified were already open issues. Although most of the code is well written, there are some functions that have not been implemented yet and are crucial to the overall development of the project. Even small bugs or errors in logic can have disastrous consequences. The code seems to undermine the importance of documentation for various operating systems.
The entire module is built in Python and hence underlying Python issues should also be identified and mitigated as soon as possible to allow widespread use of the module. While working on this OSS assessment, we realized the importance of unit testing. We have assessed the important issues associated with the module and done a comprehensive analysis of the entire code part by part. Our familiarity with Python as a programming language was a huge plus point while working on this assessment. We intend to continue working on this assessment and also use the knowledge gained in this assignment for future OSS assessments of other applications and modules.

Future work

- In the near future we intend to address most issues in marrow mailer, especially those which have been identified but otherwise not dealt with for years.
- We intend to try reporting as many bugs as possible to the author and become active contributors of the project.
- A lot of the MTAs that marrow mailer uses are outdated. Software version checks should be implemented and dependencies should be reduced, for example PyDNS dependency.

References

https://github.com/marrow/mailer

https://www.keycdn.com/support/what-is-mime-sniffing/

www.wikipedia.org

https://en.wikiversity.org/wiki/Wireshark/SMTP

https://www.zixcorp.com/resources/blog/june-2017/the-challenges-and-benefits-of-smime

https://www.cvedetails.com/vulnerability-list/vendor_id-31/Sendmail.html

https://securityaffairs.co/wordpress/66682/hacking/robot-attack.html

https://github.com/boto/boto/search?q=ses&type=Issues&utf8=%E2%9C%93

https://www.pylint.org/

https://github.com/pyupio/safety-db