

## Software Rubric Self- Evaluation

### 1. How do I meet my design specifications?

Design Criteria	Implementation Status
Working SPM Mechanism	SPM-Like Security Mechanism
Fast Authentication Protocol	Orthogonal Binary Message Protocol
Support Many Simultaneous Connections	Up to 512 limited by Asyncio Library
Secure against Chosen-Plaintext Attackers	Yes, Proof has been Provided
Secure against Unauthenticated Attackers	Yes, Attackers May Only Authenticate
Uses less than 15MB for 512 connections	Yes, Memory Scales only by Requests
Secure against Timing Attacks	Theoretically Flat Message Crypto Timing
Included Client Library	Yes, All Client Features Exposed
Administrative Management Console	Yes, Implemented Command Interpreter

Table 1: Relating specification criteria to how those specification criteria instantiate the final application.

### 2. Does my program display output correctly and intuitively?

Yes, absolutely! I have provided an interactive remote terminal, Spicy, to clearly demonstrate the behavior of the daemon. Now, the server is a daemon, and so it produces no human-readable output during normal operation. I have written a terminal to show the server behavior.

```
[pegasus2:PySPMd james$ python3 spicy.py
I'm main!
Spicy PySPMd interpreter. Type 'help' for a list of commands.
[(spicy-spm-client[~/PySPMd]) help

Documented commands (type help <topic>):
=====
auth  clearlinks  get    lcd      lpwd    mkdir   open   quit    rmsub
bye   close         gt     list_subjects  lrm     mkfilt  put    rm      shell
cd    exit          help   ll       ls      mksub   pwd    rmfilt  tt

[(spicy-spm-client[~/PySPMd]) help open
[open server port] Open a new PySPMd connection
[(spicy-spm-client[~/PySPMd]) open 127.0.0.1 5154
Successfully opened a new unauthenticated connection.
Connection established.
[(spicy-spm-client[~/PySPMd]) help auth
[auth subject password] authenticate with the connected server
[(spicy-spm-client[~/PySPMd]) auth admin password
Authenticating...
Authentication success.
[(spicy-spm-client[~/PySPMd]) list_subjects
Available Subjects:
admin
[(spicy-spm-client[~/PySPMd]) put test.bin
[(spicy-spm-client[~/PySPMd]) ls
Available Objects:
test.bin
[(spicy-spm-client[~/PySPMd]) rm test.bin
[(spicy-spm-client[~/PySPMd]) ]
```

Figure 1: The Spicy remote command interpreter for PySPM is the primary user-visible interface to the daemon. The daemon itself relies on a binary message format used to communicate with applications.

Correct and intuitive output is less relevant of a condition for applications that have no interface, such as my PySPM daemon. Nevertheless, I have provided such an interface to the daemon for demonstration purposes. The “help” command provides full documentation for the interpreter and each of its commands. The underlying binary messaging system works correctly, because without it, such an interpreter could not operate correctly either.

### 3. Is my code readable?

My code is written almost exclusively in Python 3. Python is known for its extreme readability, and many programmers consider Python a kind of executable pseudocode. My code contains plenty of comments and has few dependencies, depending only on the standard library, which leads to both code portability as well as high readability among Python 3 programmers.

```
def do_list_subjects(self,line):
    """[list_subjects] list all valid subjects on the server"""
    if not self.client or not self.client.connected:
        print("No active connection")
    else:
        subjects = self.client.listSubjects()
        print("Available Subjects:")
        [print(subject) for subject in subjects]
```

*Figure 2: The PySPM source code contains numerous inline comments and documentation strings. This source code is highly readable even to non-programmers, owing primarily to the intuitive and simple syntax of Python.*

Python does not use header files, however, the language provides the doc-string inline documentation format. Here, a doc-string indicates this method lists all the valid subjects on the server. The documentation string for any function in my code can easily be accessed through the following convention:

```
>>> import SPM
>>> import SPM.Client
>>> help('SPM.Client')
```

Continued on next page...

```

Help on module SPM.Client in SPM:

NAME
  SPM.Client

CLASSES
  builtins.RuntimeError(builtins.Exception)
      ClientError
  builtins.object
      Client

  class Client(builtins.object)
      | Client library interface object
      |
      | Methods defined here:
      |
      | __init__(self, addr, port)
      |     Initialize self.  See help(type(self)) for accurate signature.
      |
      | authenticate(self, subject, password)
      |     Authenticate as a subject and establish encryption
      |
      | cd(self, remotepath)
      |     Change virtual remote path on the server
      |
      :

```

Figure 3: Using the built-in documentation viewer. Documentation for PySPM is automatically generated using the Python 3 documentation string interface. The documentation for any class, method, or module can be accessed using `help('Entity')` in the interactive Python 3 interpreter.

Every component of the code is documented. The documentation for each code fragment can be accessed interactively using the standard library documentation interface as illustrated in Figure 3.

#### 4. Is the code and all its routines reusable?

Yes, my code is highly and immediately reusable as presented. There is no need to load the entire module to access a single function. For example:

```

>>> from SPM.Client import Client
>>> leave_server = Client.leaveServer

```

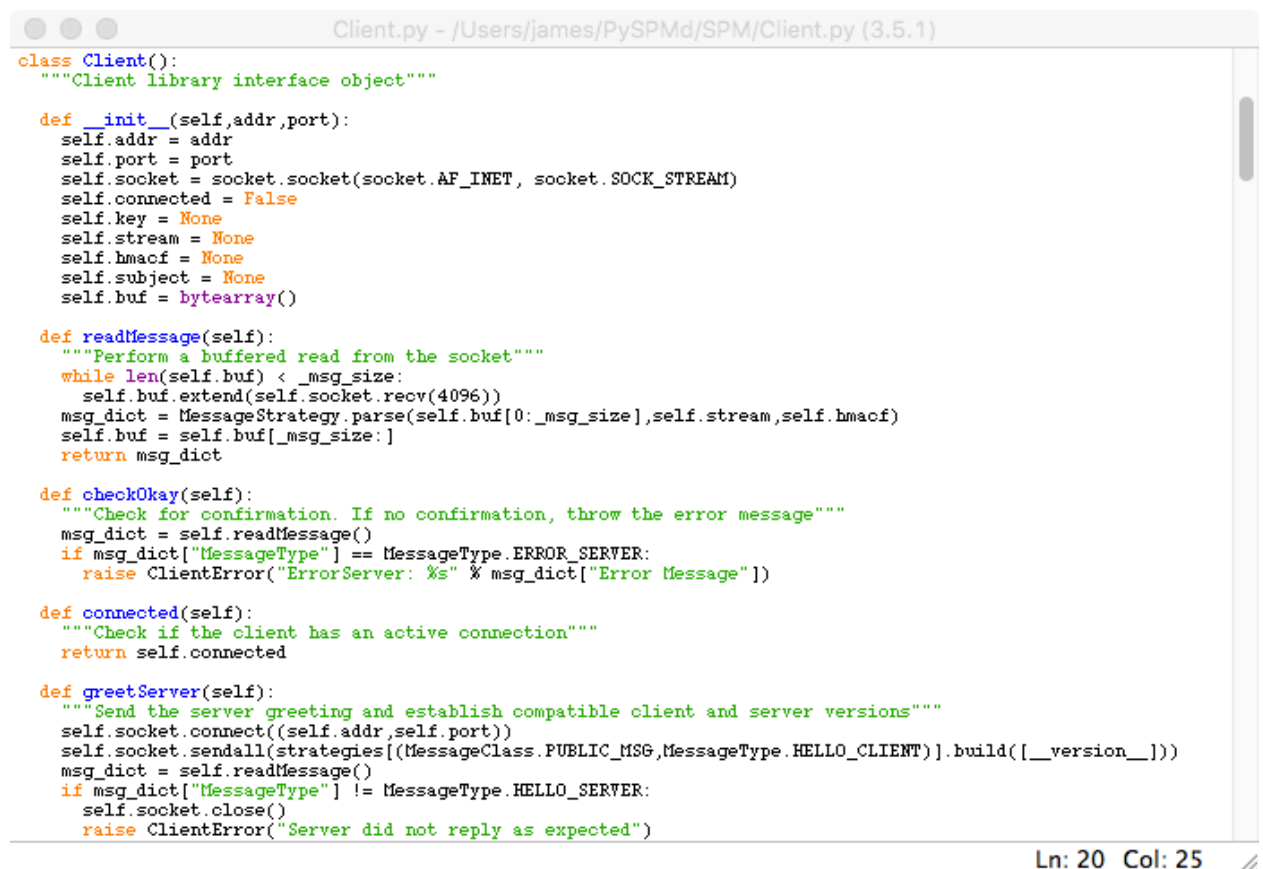
This code will extract the Client interface from the SPM module, dynamically unbind the method from the class, and extract it as the procedure 'leave\_server'. Even the documentation for each function is preserved because the documentation is bound to each block of code, not the project as a whole:

```
[>>> from SPM.Client import Client
[>>> leave_server = Client.leaveServer
[>>> leave_server
<function Client.leaveServer at 0x1d3a26c>
>>> help(leave_server)]
Help on function leaveServer in module SPM.Client:

leaveServer(self)
    Disconnect from the server, preparing for any future connections
(END)
```

Figure 4: Illustrating the highly modular and flexible nature of the code and its documentation. Here, a single method is extracted from `SPM.Client`. The method documentation automatically follows the function to wherever the procedure will be used.

The PySPM project is organized into modules, which are further organized into classes and methods of classes using a strong object-oriented programming model. Each bound method and object from the program can be instantly reused in other projects, and furthermore, the documentation for each unit of code follows the exported routines automatically.



```
class Client():
    """Client library interface object"""

    def __init__(self, addr, port):
        self.addr = addr
        self.port = port
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.connected = False
        self.key = None
        self.stream = None
        self.hmacf = None
        self.subject = None
        self.buf = bytearray()

    def readMessage(self):
        """Perform a buffered read from the socket"""
        while len(self.buf) < _msg_size:
            self.buf.extend(self.socket.recv(4096))
        msg_dict = MessageStrategy.parse(self.buf[0:_msg_size], self.stream, self.hmacf)
        self.buf = self.buf[_msg_size:]
        return msg_dict

    def checkOkay(self):
        """Check for confirmation. If no confirmation, throw the error message"""
        msg_dict = self.readMessage()
        if msg_dict["MessageType"] == MessageType.ERROR_SERVER:
            raise ClientError("ErrorServer: %s" % msg_dict["Error Message"])

    def connected(self):
        """Check if the client has an active connection"""
        return self.connected

    def greetServer(self):
        """Send the server greeting and establish compatible client and server versions"""
        self.socket.connect((self.addr, self.port))
        self.socket.sendall(strategies[(MessageClass.PUBLIC_MSG, MessageType.HELLO_CLIENT)].build([__version__]))
        msg_dict = self.readMessage()
        if msg_dict["MessageType"] != MessageType.HELLO_SERVER:
            self.socket.close()
            raise ClientError("Server did not reply as expected")
```

Ln: 20 Col: 25

Figure 5: The PySPM project uses a class-based system to organize code and data. A side effect of this class hierarchy is convenient code and object re-use.

5. Is the user documentation clear and well-written?

Yes, and furthermore, examples on the proper use of the module are provided. Each method serves the user with auto-generated documentation, which can also be extracted in its entirety through object introspection. The project design philosophy centers around keeping each object and method as small and as understandable as possible. This paradigm maximizes the potential for code reuse and makes the code easier to understand. If a method requires more than one sentence to describe its use and operation, the method was fragmented such that each method only implements a singly, simple, easily understandable and testable functionality. Users browse the documentation interactively using the `help()` interface of the Python 3 standard library.

Each method should do one thing, one thing only, and one thing well.

6. The theory of operation is explained thoroughly and clearly.

As this document is part of the user's documentation, let us review the essential theory of operation right here. Additional overviews of the object operation are available in the thumbnail presentation and project proposal documents.

This program is an asynchronous non-blocking socket server implementing the SPM protection model. The program operation is built around a central event loop, the main loop. When there is nothing to do, the server remains blocked on the event loop, waiting for events to be generated. The event loop blocks on the `select()` system call, waiting on activity from a client socket. When a client connects and requests operations such as transfer of rights, authentication, or access to data, each request is parsed using the `Message` class. Each message block received maps to an event which is then injected into the main loop. The main thread executes this event, a coroutine, optimistically at its earliest convenience: for example, while another client instance is blocked performing IO operations. This design ensures that the main interpreter thread reaches maximum utilization and never waits on a client at any time while there is other work to do. Every event maps to a coroutine, which calls subfunctions and schedules other coroutines to execute where an ordinary threaded socket server would have blocked.

Program Flow of Main Thread

Block in main loop on event queue

Pop event from the queue

Execute event's corresponding coroutine in the context of the client protocol (state) object until execution yields

### Program Flow of a Client Request

Client makes request

Select() system call unblocks the main thread if it is sleeping

The socket is used to look up the Client protocol object, holding the client state

Client message is encapsulated in an event coroutine

Coroutine is scheduled to execute with event data and client state object

Main thread executes coroutine at earliest convenience (such as when a worker thread in the traditional socket server model would block)

The execution model uses reactive programming and relies on the concept of the continuation. For more information about Python 3 coroutines, see the asynchronous coroutine proposal and request for comments [PEP-3156](#).

### Advantages of Reactive Model

Only a single, main thread

Each coroutine executes atomically

Locking is replaced by coroutine scheduling

Each additional client does not require his own call stack

No overhead of thread destruction and creation

Not limited by operating system multitasking restrictions

This approach to server programming is relatively new, and has not been taught in University of Florida courses based on my limited experience. Although not the obvious way to implement a server taught in classes, it is highly efficient because it transcends the difficulties of implementing correct multithreading. This design pattern is used in the popular Node.js asynchronous JavaScript server implementation.

7. Test cases are thorough and systematic. Proofs of correctness are supplied or cited.

The automated test harness launches from Spicy and the TestServer/TestClient combination. The Proofs of correctness are academically most interesting, and so the lions share of this point is expended on relevant proofs of correctness.

From the Project Update:

The design of the network protocol and client-server model was created with great care to ensure ease of implementation and security of the protocol. Firstly, Python performs all message parsing to protect against attacks on the memory-safety of the parser. Second, the design of the network protocol is purely orthogonal. All messages have the same format, are the same length, and will appear as random bit strings of the same length to an attacker. The authentication tag prevents message modification in transit. A secure stream cipher (marked with repeating XOR) is exclusive-or'd with the message, the key based on a shared secret and a nonce chosen by the client in the connection. All together, the network architecture is simple and orthogonal, providing privacy,

authenticity, orthogonally, and easy of implementation. A proof of cryptographic security follows easily from the entropy of the stream cipher called PRF-security, and will be provided in full in the accompanying paper. In brief, the distribution of the encrypted messages can be shown to be random and uniform of the stream bits are uniform and identically randomly distributed.

Let us define the following cryptographic game called the ciphertext indistinguishability game on pseudorandom permutation E

PRF-A

$K \leftarrow \{0,1\}^n$

$b \leftarrow \{0,1\}$

$b' \leftarrow A^{O(\cdot)}$

*Ret*  $[b' = b]$

Oracle O(p)

If  $b = 1$

*Ret*  $Perm(l)$

else

*Ret*  $E_K(p)$

Let us define the advantage of an adversary A as:

$$Adv_E^{PRP}(A) = 2 * \Pr[Exp_R^{PRP}(A) = 1] - 1$$

Where “n” is the key size, “l” is the block length, and “b” is a challenge bit selecting whether or not cipher E or a random permutation of length “l” is implemented in the oracle. The adversary A wins the PRP game if it can distinguish with a positive advantage between the cipher and random bits.

Provided that the output of E is sufficiently random, it is indistinguishable for a random permutation. However, a more desirable property of this cryptosystem is that an adversary should not be able to recover the key. Now consider the following security reduction:

$$\neg KeyRecovery \Rightarrow \neg PRP$$

In other words, if it is true that a key recovery attack implies that E does not implement a pseudorandom permutation over bit strings of length “l,” then it follows that resistance in the PRP game is a sufficient condition to prevent recovery of the secret key.

Let us reduce the PRP adversary B that efficiently performs key recovery into an adversary A that efficiently distinguishes the output of E from a random permutation.

KeyRecovery B

$K \leftarrow \{0,1\}^n$

$K' \leftarrow B^{O(\cdot)}$

*Ret*  $[K' = K]$

Oracle  $O(p)$

*Ret*  $E_K(p)$

Adversary PRF  $A_B$

$b \leftarrow \{0,1\}$

Run B

When B asks  $O(p)$

$Y \leftarrow O(p)$

Return Y to B

When B returns with output  $K'$

*Ret*  $[E_{K'}^{-1}(Y) = p]$

Where “p” is any query made by B to the oracle. Clearly, an adversary that performs key recovery enables an adversary that can efficiently distinguish between E and a random permutation. The advantage of  $A_B$  will be the same as B because whenever B wins the KeyRecovery experiment,  $A_B$  wins the PRF experiment, plus a small term for when the random permutation incidentally decrypts under key  $K'$  to “p.” Thus,  $\neg \text{KeyRecovery} = > \neg \text{PRP}$  and  $\text{PRP} \Rightarrow \text{KeyRecovery}$ . PRP is a sufficient condition to prevent recovery of the secret key. Intuitively (and we have proven) that a truly random bit string reveals nothing about the secret key. If we believe that the stream cipher represents a pseudorandom permutation (and is indistinguishable from a truly random permutation), then key recovery is not possible and the network protocol is provably secure.

An additional proof for HMAC authenticity exists. Because I use a standard SHA1-HMAC, I will defer the proof of authenticity to [a public peer-reviewed document](#) in my paper. Confidentiality and integrity are related but one is not a function of the other.

## 8. Is the code efficient?

Now, before answering this question, think carefully. What does it mean to be efficient? This very computer uses a Von Neumann architecture which is well-known to be highly inefficient for applications such as neural network simulation. I define efficiency as follows:



*An efficient program is a program where the ratio of provider costs vs. consumer utility is maximized.*

I choose this definition because it reflects efficiency in practice. A program that costs one million dollars a day to maintain and is only slightly better from a user perspective than a program that costs nothing to use each day is obviously a bad program. Execution efficiency is only one element of the cost to provide the service. In reality, efficiency of a solution depends on many factors. Developer time is far more expensive than CPU time. You can always buy more server as long as the benefits of an additional server outweigh the costs.

PySPMd is efficient because it is easy to read, maintain, extend, and is highly scalable. Its latency is deterministic in that it does not rely on a garbage collector. Although it is written in Python 3, the program is secure against the most common mistakes that programmers make. I know I have introduced no buffer overflows or resource leaks. The code is well-documented and performant in the problem domain in which it performs. In this sense, the code is efficient, even more efficient than a similar network-facing C++ program that requires far more error-prone man hours to maintain plus additional security risks by virtue of this memory-unsafe language.

## 9. Data structure performance analysis.

This project is a network application that indirectly exposes access to a data store. Thus, most of the performance considerations of the application are determined by the data structures used to store and process network data. The in-process database, Sqlite3, uses B-trees to efficiently process queries. I have not implemented this data structure myself, but I trust this widely-used library to implement it correctly. Other data structures include the closure, the hash table, and the event queue that supports fast pops and appends from both ends of the queue. I have not implemented any complex data structures from language primitives. My project uses design-by-contract, relying on contracts with other components that provide features such as fast pops and appends without specifying the underlying implementation.

I have implemented a network protocol with a large list of possible message types. In most messages that do not transfer files, the vast majority of the message block is wasted. This is very inefficient, but it is the only way to protect against statistical packet analysis based on the length of the message. If messages had different sizes, an attacker could distinguish the messages. The earlier proof relies on the assumption that the message length reveals nothing about the message content. In order to build on this assumption, all messages are of equal length; each message is indistinguishable from a 2048-byte block of random data. At a minimum, each message has an overhead of 22 bytes (two bytes for the message class and header, and twenty bytes for the message authentication and integrity tag). The greeting and confirmation messages are the worst

offenders, with the entire message consisting only of the message class, type, padding, and authentication tag. The network protocol must be secure. This imposes a necessary restriction on the efficiency of the protocol. In other respects, each message is of length 2048-bytes and has a trivial performance analysis of 2048-bytes to communicate each message structure as defined in the Message class.

## 10. Security Considerations

### a. Buffer Overflow and Memory Management Attacks

Not possible as long as Python implementation is secure. The language does not expose the necessary primitives for parsing to cause an unexpected loss of program flow control. I cannot have made an error that leads to a buffer overflow attack.

### b. Denial of Service Attacks

Denial of service is an unsolved problem in computer security. Any sufficiently powerful attacker can deny service by flooding the network connection. However, the program is effective against this security consideration because every attacker is limited only to authentication before the user has authenticated, and while authenticating, the attacker cannot trigger a large amount of work to happen on the server in exchange for a comparatively small amount of work on the attacker's part. Finally, simple rate-limiting is implemented. A login delay prevents attackers from quickly implementing dictionary attacks using a single connection. Limiting the number of connections can also be done at the system firewall level.

### c. Network Sniffing Attacks

All messages are encrypted using RC4-DROP-2048, which has no known distinguishing attacks in the current public literature. When logging in, a client submits a username with a random 32-byte salt value. The password in any form is never transmitted over the clear, hashed or otherwise. The ciphertext is a function of the clear-text message, the salt (known to the attacker) and a shared secret password that is never transmitted. A key derivation function derives the session key from the salt and the shared secret password. If no information can be gathered from the cipher about the key (we assume no weaknesses in the cipher), then the password cannot be recovered, nor can any plaintext be recovered, from a network sniffing attack. PKCS7 is used as the session key derivation function. If we accept that RC4-DROP-2048 produces random uniform bits and that the key derivation function does not leak information about the shared secret, then network sniffing cannot reveal the password or any information about the message content. A related proof has been presented.

d. Timing Attacks

A timing attack uses timing information in the server responses to determine information about secret the state of the system. To prevent timing attacks, a special [constant-time HMAC comparison function](#) is used that has a flat timing profile to hide the incremental process of the string comparison. Additionally, authentication attempts are clouded by a randomized delay in addition to the login rate-limiting. Authentication failures do not indicate what part of the information failed to authenticate, returning generic failure even for an invalid username (subject) in this application.

e. Message Forgery

Each message is protected using HMAC-SHA1, a popular and widely used message authentication code that is parameterized by the session key. The HMAC is used to sign the message ciphertext, so it cannot leak information about the message content. If the ciphertext or its authentication code are modified in any way, the attacker cannot hope to forge a valid authentication code. Messages without valid authentication codes are ignored.

[HMAC Proof of Security.](#)

Complete references are provided in the associated paper, placed in the same directory as this document upon its completion.