James B. Birdsong
Professor Newman
Information Security
9 April 2016

PySPMd: Access Control Daemon

Introduction

North Korea hacks Sony and steals sensitive company assets. Contractor Edward Snowden leaks highly-compromising, classified government documents to the general public and enemies of the United States. The more recent Apple iCloud celebrity account compromises disclosed intimate details of the private lives of popular Apple iCloud users. What do all of these unfortunate events have in common for each respective organization? Each of these events extends naturally from a failed access control mechanism. In the case of the Sony hack, attackers accessed assets, company information, and pre-release films without authorization. Snowden collected classified documents because he had access to information greater than what was necessary to perform his duties. For Apple, iCloud password recovery mechanisms leaked the right to account access to unauthorized parties. Poor access control is a serious problem in the technology sector, with numerous profound and far-reaching consequences in the public eye ("Reputation Impact"). Clearly, the technology industry has much to learn to adequately protect information security and to enforce appropriate security policies ("40% of Security Experts"). In pursuit of the need for authenticated, secure, and centralized access control mechanisms, the PySPMd access control system implements a fast and effective access control security mechanism that limits access to information to authorized entities according to the rules of the Schematic Protection Model.

Background

From the beginning, access control has always played a critical role in maintaining the confidentiality, integrity, and accessibility of resources. The agricultural revolution converted late nomadic hunter-gatherer societies into a more persistent form, reliant on perpetuating control over long-term assets. In particular, early peoples built lasting abodes and with these changes came a need for physical access control to secure their bustling homesteads. In 2016, controlling physical access to resources remains equally as important as it did in early agricultural societies. The barter system yields to a postmodern global financial system, bank accounts, shopping malls, and investment firms. The abstraction of value paradoxically demands limited access to these abstractions before, in computing, limitless demands for physical memory pages preclude page faults and allocation failures in virtual memory systems, and while Walmart supercenters deal in all manner of goods and services, a small underclass of global citizens suffers and dies in the streets (Martin). Physical access control partitions the global economy into haves and have-nots, and yet without this necessary evil, post-modernism, modernism, industrialization, and indeed the foundations of a society reliant on its agricultural underpinnings fracture where finite supply meets virtually infinite demand.

While information goods differ from physical goods in several important ways, access to information commands as much as, if not more, influence. Information goods, or content, differs in that the marginal cost of production for the next unit of information approaches zero. While contractors require increased access to resources to instantiate each additional instance of physical infrastructure, the marginal cost to produce additional licensed copies of extant software costs next to nothing by comparison. In the technology sector, software products have a high initial development cost that producers amortize over sales, where each additional license costs no additional investment from the producer except perhaps a small amount of cheap bandwidth and storage. The Internet represents the largest, most efficient copying machine in the history of the world. From this, analysis might conclude that information does not require access controls. If information is cheap, then why should anyone bother with access control? The answer lies in the high and increasing costs of unauthorized access to sensitive information (Whittaker). High initial investments demand strict access control measures to extract a profit. Records, designs, methods, and other secrets are secret because they extract very real harm upon unauthorized access. Information access controls are no less important, and recent failures of information access control mechanisms highlight the very real value of information content.

Access control is important, and this is true for both physical and information access control mechanisms. Nevertheless, correct and effective access control remains a hard problem. In a perfect world where access control correctly enforces all security policy, no credit card theft, malware infections, identity theft, or leaked documents could exist. However, in the real world, all of these aspects of problematic information security accurately describe the state of the art. Physical access control mechanisms can be broken. Locks are picked, safes are opened, identification cards are stolen, and resultantly, unauthorized parties access physical resources to which these parties are not authorized. While physical access mechanisms can always be broken with sufficient physical resources, information access control mechanisms most often fail to logical errors in their implementations. In the case of Apple iCloud, the implementation fails to correctly limit authentication attempts to reasonable parameters (Donovan). Contributing further to the difficulty, the question of safety of whether a right can be leaked to an unauthorized party is undecidable in the general case for the access control matrix model (“Foundational Results”). An arbitrary Turing machine can be reduced to the safety question using the access control matrix model. Only for a restricted subset of protection mechanisms have answerable safety questions, complicating the security assessment of an information access control mechanism. Access control is a hard problem, and while we cannot hope to build secure systems in the general case, careful design can lead to effective mechanisms implementing security policy.

Implementation

PySPMd is a centralized, concurrent, asynchronous, non-blocking, networked information access-control daemon storing objects and serving access to objects according to the schematic protection model (Sandhu). Subjects authenticate with the server and make requests on behalf of the subject, and the server provides data access and performs the requested operations according to the security policy. Internally, the server uses reactive programming with an event loop and an in-process database to store the state of the model. Objects are stored as files on disk. PySPMd is implemented using state-of-the-art design patterns to maximize maintainability and portability across systems. The program is implemented in a memory-safe language, protecting against the most common network threats. Several classes of attacks were considered and mitigated at the conclusion of the design process. This section details the implementation of PySPMd, including the design motivations, considerations, and processes used by the daemon to achieve its ends.

While the access control problem poses its own unique challenges, we often overlook the importance of an underlying problem: How do we implement secure, centralized, network-based services? Perhaps the oldest form of multiprogramming included the use of multiple programs communicating by shared access to configuration files. As multiprocess and timesharing systems matured, the first consumer-producer problem emerged, and processes began to communicate. Processes run in separate address spaces and do not generally perform inter-process communication by default. A common design pattern is for a server to accept connections, fork() to establish a new process, and then proceed to handle the client connection with each process handling a single client. However, this approach is memory-intensive, especially on systems that do not offer copy-on-write semantics, and sharing of large, complex data structures requires careful communication between processes to maintain the order of the shared state. Another approach is to utilize multiple threads of execution within the same process address space. However, threads are evil (Lee) in that threading introduces an entirely new class of particularly hard to diagnose concurrency problems. Threaded programs are non-deterministic in the precise order of instruction execution. Writing a multi-threaded program requires attention to detail and data protection awareness to write correct programs, and experience shows that threaded application tend to contain difficult-to-expose bugs. Regardless of whether process or thread-based concurrency primitives support the application, each introduces challenging questions and raises the bar on the programmer.

In the design of PySPMd, there is only a single process and a single thread. No locking procedures are used, except for coroutine synchronization and even then this locking is used only as a performance optimization. No complex data structures requiring special analysis are used. PySPMd concurrency espouses reactive, event-driven network programing where coroutines cooperatively share the main thread as it executes the coroutines in the program's main loop. Client requests directly map to coroutines, each scheduled for execution in the reactor (main loop). Where a thread would normally block, a coroutine yields execution until the requested resources are available. This keeps the main thread busy as long as there is computing to do. The main thread blocks on the select() system call when all coroutines are waiting for data from a network socket. Each coroutine executes atomically, and every client request requires only a small stack frame rather than its own stack space for a thread. The design of PySPMd enables large-scale concurrency and large-scale applications by demanding less from the programmer and less from the underlying system to implement its input-output concurrency design.

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

From a networking perspective, PySPMd presents an extremely simple analysis. All messages are exactly 2048 bytes long regardless of content. The first two bytes identify the message type, while the last twenty bytes form the message authentication code which is itself the output of a pseudorandom function. The message type defines the internal structure of the 2048-byte message block. Before PySPMd sends a message, the message is first encrypted using AES-256-CTR or RC4-DROP-2048 depending on the platform. Key negotiation uses a shared-secret key and a 32-byte random number chosen as the initialization vector by the client. The authentication tag is appended. Special timing-invariant functions implement message authentication and encryption to resist timing attacks. The leading byte, the message class field, indicates whether or not the rest of the message is encrypted. When the message is decrypted, its type can be determined and the client or server may respond appropriately. Most messages contain a large amount of unused space. In this sense, the protocol is wasteful of bandwidth, but a compromise is necessary to protect length information that could otherwise discriminate between activities undertaken by the connected client.

In brief, the implementation of PySPMd is one of a socket server with an attached SQL database to store client information. In this respect, the implementation of PySPMd is not unique. What is unique about the design of this information security protection daemon is its unique approach to centralized resource access control. The schematic protection model describes the data model while reactive design describes the control flow that implements the simple orthogonal message protocol. A simple encrypted network protocol hides message content as well as content length with a large bandwidth cost for most small operations. PySPMd provides a highly concurrent network interface to data, moderated by an SPM-based access control mechanism.

Security Considerations

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
$K \leftarrow \{0,1\}^n$
$b \leftarrow \{0,1\}$
$b' \leftarrow A^{O(.)}$
$Ret\ [b' = b]$

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:

$$\neg KeyRecovery => \neg PRP$$

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
$K \leftarrow \{0,1\}^n$
$K' \leftarrow B^{O(.)}$
$Ret\ [K' = K]$

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, $\neg KeyRecovery => \neg PRP$ and $PRP => KeyRecovery$. 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. Proofs of authentication security using HMAC-SHA1 are widely available (Krawczyk). The above proof is reproduced from the Project Update document.

Weakness and Remediation

- Buffer Overflow and Memory Management Attacks

  This class of attacks is not possible as long as the 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.

- 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.

- Network Sniffing Attacks

  All messages are encrypted using RC4-DROP-2048 or AES-256-CTR, 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. PKCS is used as the session key derivation function (Kaliski). If we accept that RC4-DROP-2048 or AES-256-CTR 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.

- 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.

- 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.

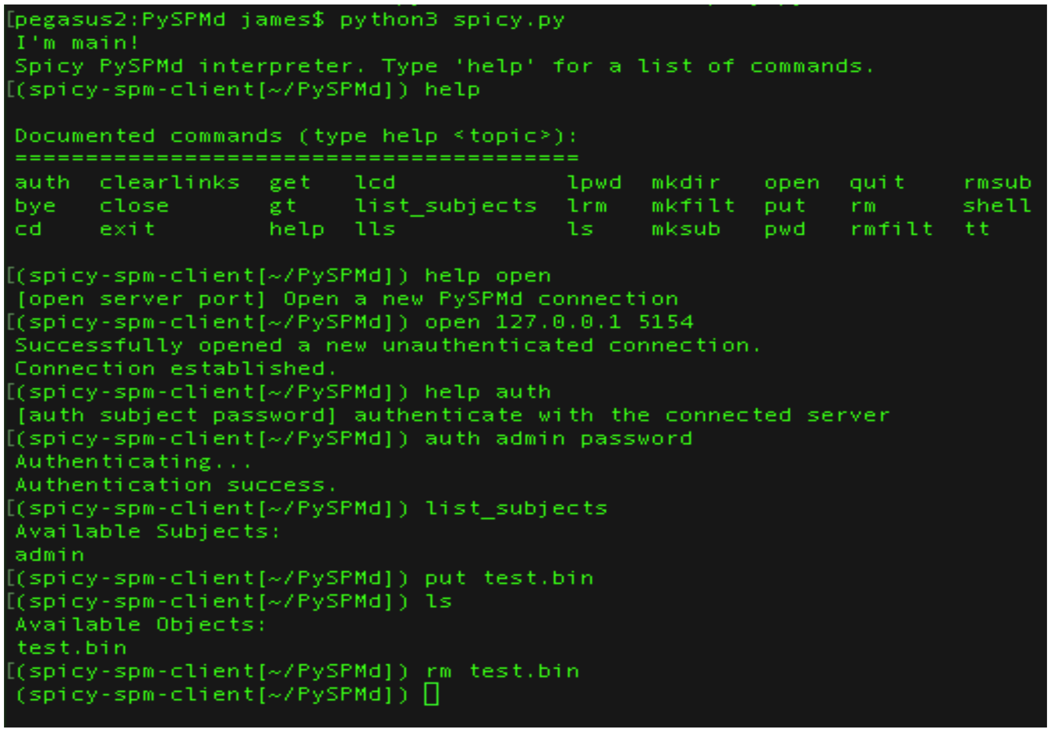
Conclusion

PySPMd is a versatile and secure information security protection daemon, serving resources to authenticated users. The background, motivation, design, and security considerations of the software constitute this paper as well as it constitutes the relevant software documentation. By no means is this daemon an end-all, be-all, turn-key solution to the problems plaguing the information technology sector today. The daemon was written first and foremost to illustrate concepts and encourage organized, structured thought about how such a protection daemon should be written. This software is provided in the hopes that it will be useful to someone else as a pedagogical learning tool, illustrating concepts in reactive programming and information security access control.

Selected Illustrations from Program Documentation

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.

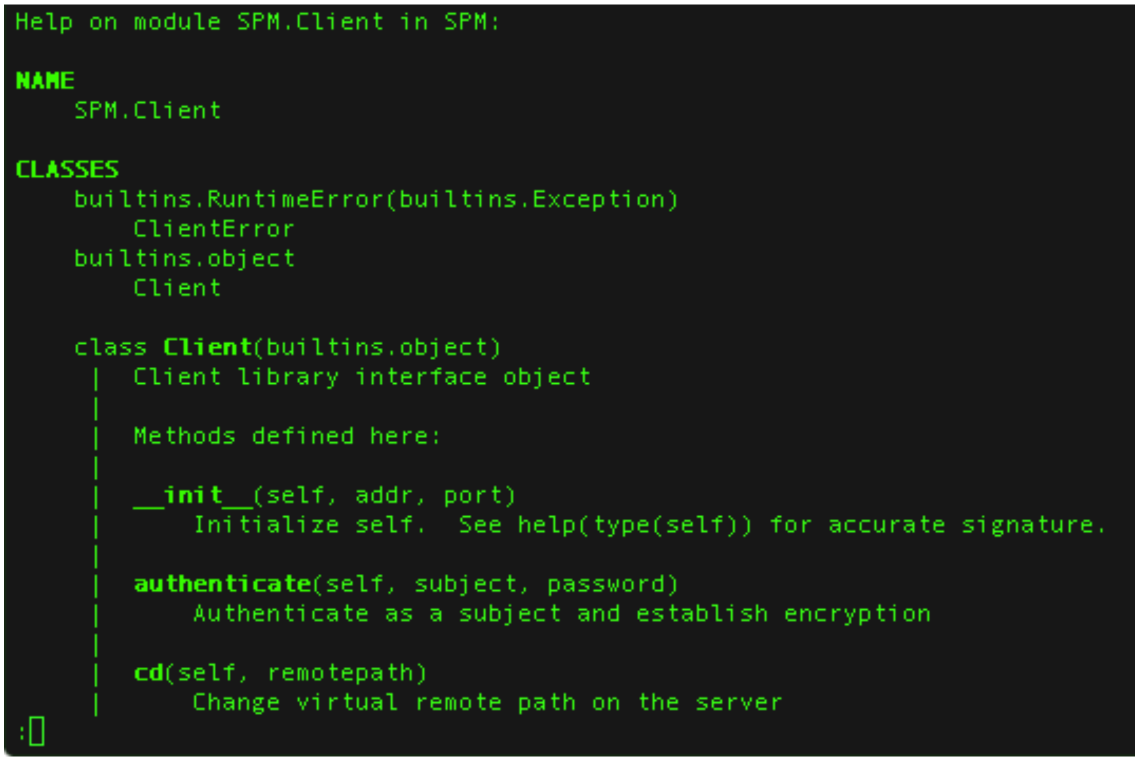


```
[pegasus2:PySPMd james$ python3 spicy.py
 I'm main!
 Spicy PySPMd interpreter. Type 'help' for a list of commands.
[(spicy-spm-client[~/PySPMd]) help

 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  lls            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 PySPMd is the primary user-visible interface to the daemon. The daemon itself relies on a binary message format used to communicate with applications.



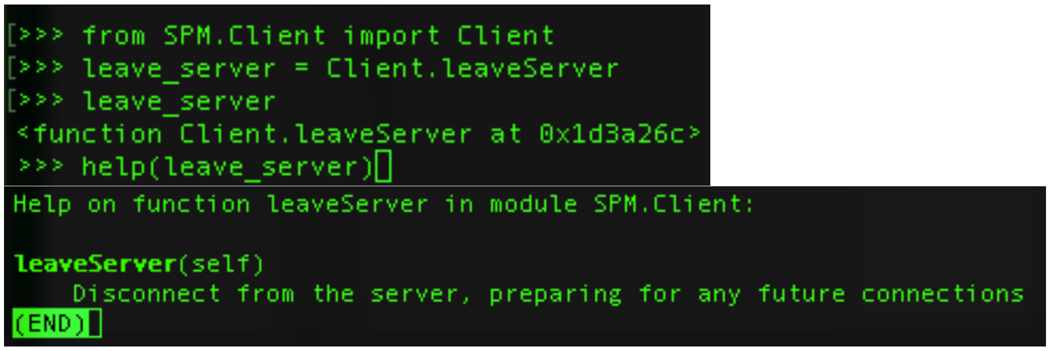
```
Help on module SPM.Client in SPM:

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
:
```

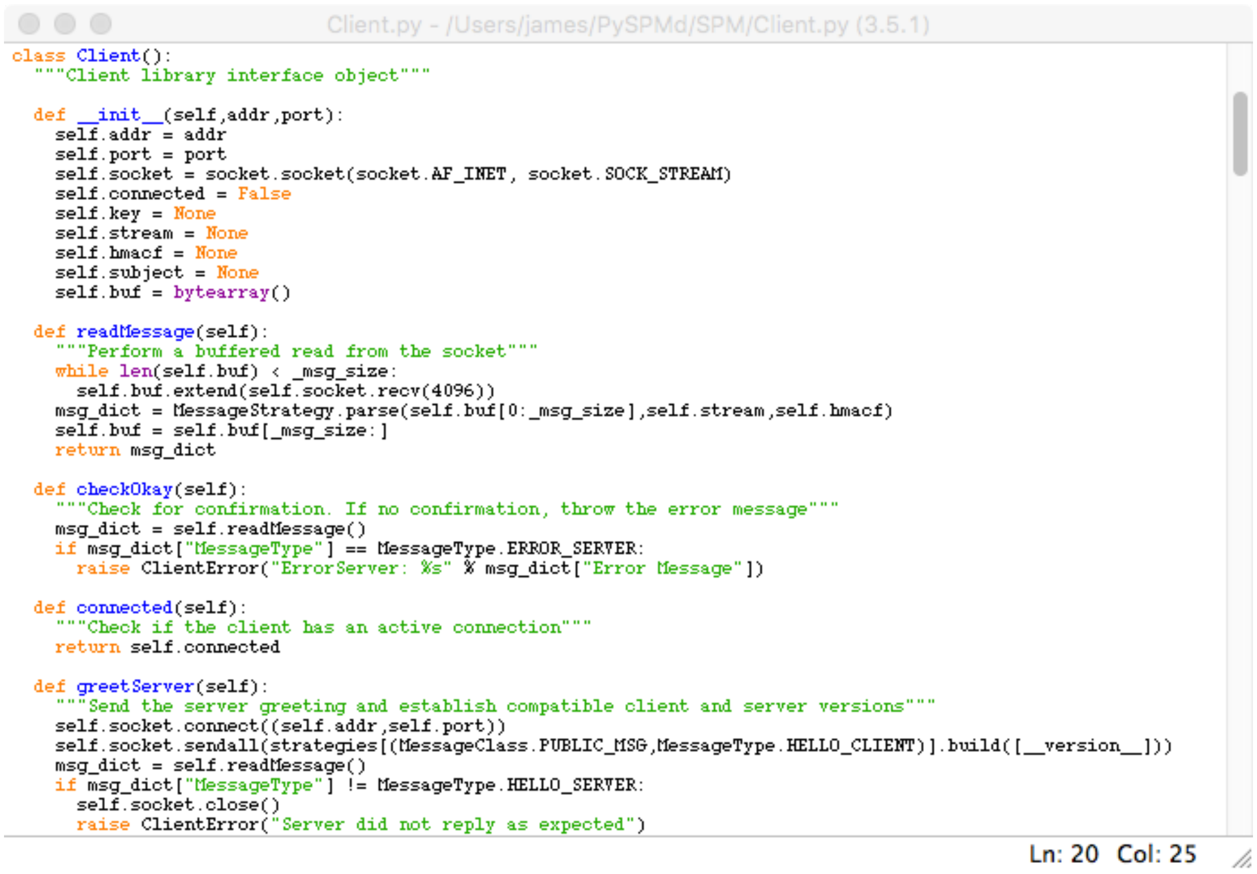
Figure 2: Using the built-in documentation viewer. Documentation for PySPM is automatically generated using the Python 3 documentation string (help()) interface. The documentation for any class, method, or module can be accessed using help('Entity') in the interactive Python 3 interpreter.



```
[>>> 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 3: 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.



```
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")
```

Figure 4: 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.

Works Cited

Bellare, Mihir. "New Proofs for NMAC and HMAC: Security without Collision-Resistance." (n.d.): n. pag. June 2006. Web. 11 Apr. 2016.

Donovan, Fred. "Apple's ICloud Breach: It's Not Just About Naked Photos." *FierceITSecurity*. FierceMarkets, 2 Sept. 2014. Web. 11 Apr. 2016.

"40% of Security Experts Predict Insider Data Breaches in Coming Year." *SC Magazine UK*. Ed. Danielle Correa. Haymarket Media, 18 Nov. 2015. Web. 11 Apr. 2016.

"Foundational Results." *Auburn Department of Engineering*. Auburn University, n.d. Web. 11 Apr. 2016. <http://www.eng.auburn.edu/~weishinn/CSCI0421/Chapter%203.pdf>.

Kaliski, B. "PKCS #5: Password-Based Cryptography Specification." *IETF*. The Internet Society, Sept. 2000. Web. 26 Feb. 2016.

Krawczyk, H., M. Bellare, and R. Canetti. "HMAC: Keyed-Hashing for Message Authentication." *IETF*. The Internet Society, Feb. 1997. Web. 11 Mar. 2016.

Lee, Edward A. "The Problem with Threads." (n.d.): n. pag. Electrical Engineering and Computer Sciences University of California at Berkeley, 10 Jan. 2006. Web. 11 Mar. 2016.

Martin, Patrick. "Capitalism and Global Poverty: Two Billion Poor, One Billion Hungry." *Global Research News*. GlobalResearch, 25 July 2014. Web. 11 Apr. 2016.

"Reputation Impact of a Data Breach." (n.d.): n. pag. *Experian Information Solutions*. Experian, Nov. 2011. Web. 11 Apr. 2016.

Sandhu, Ravinderpal Singh. "The Schematic Protection Model: Its Definition and Analysis for Acyclic Attenuating Schemes." *Journal of the ACM JACM J. ACM* 35.2 (1988): 404-32. *University of Pittsburgh*. Web. 26 Feb. 2016.

Watson, Devin. "Linux Daemon Writing HOWTO." *Linux Daemon Writing HOWTO*. N.p., May 2004. Web. 26 Feb. 2016.

Whittaker, Zack. "Data Breaches to Cost $2 Trillion by 2019." *ZDNet*. CBS Interactive, 12 Mar. 2015. Web. 11 Apr. 2016.