James Birdsong
Professor Newman
Information Security
11 March 2016

Project Update: Schematic Protection MAC

The project is now well underway and has taken great shape on the grounds of network, database, and software architecture. I have taken the path less traveled at every turn to maximize the learning potential imbuing the experience. Over time, I have also learned to be more realistic about my design goals and to select those objectives that make the software most interesting and worthwhile for both users and its developer. As with most aspects of life, the project has taken longer than stated in my initial projections. Emergent issues were encountered and settled. This document should be viewed as an extension of the project proposal—it addresses the issues that have been encountered, the present software architecture from several perspectives, and deepens the background of the project to best situate the work in the context of popular mandatory and discretionary access controls available for the Unix platform.

Problems and Solutions

1. Early support for stop-gap continuations implementation in Python 3.

   Python 3.5 introduces the asyncio library, providing temporary support for continuations, event loops, and functional approaches to writing socket servers. However, one would do well to roll their own implementation. The latest implementation is provided strictly on a provisional basis, with striking backwards-incompatible changes possible at any time and without prior notice. I have chosen to implement my own simple event loop and queue depth maintenance routines at the price of some latency relative to a multiprocess or threaded implementation. In this way, I have chosen throughput and memory-efficiency over request latency.

2. Python virtual machine global interpreter lock (GIL) severely handicaps parallel interpreter thread execution.

   The Python virtual machine's automatic memory management is not thread-safe. To allow threads for IO-bound processes and as a programming model, the developers chose to implement a GIL. Any interpreter thread executing Python bytecode must hold this lock. This means that a threaded socket server can only parallelize IO operations, not computation or database accesses. I have sidestepped this issue by using an event queue with one worker thread to serialize many simultaneous events into a single event stream. Each event is handled synchronously, but many yield to other events when performing expensive operations. This provides the appearance of parallel event handling without GIL contention, which would degrade performance.

3. Processing untrusted data from the Internet using C/C++ routines is insecure.

   More often than not, complex parsing code written in memory-unsafe languages harbors buffer overflow exploits just waiting to be discovered. To ease the project implementation and enhance the security of the public-facing program components, the message parsing and protocol is implemented in Python.

4. Python is slow and imposes a large constant complexity factor on algorithms.

   Write performance-critical code sections in C while security and ease-of-use-critical sections in Python. This hybrid approach yields the best of both worlds. Amdahl's law indicates that a large speedup can be realized by optimizing critical code paths.

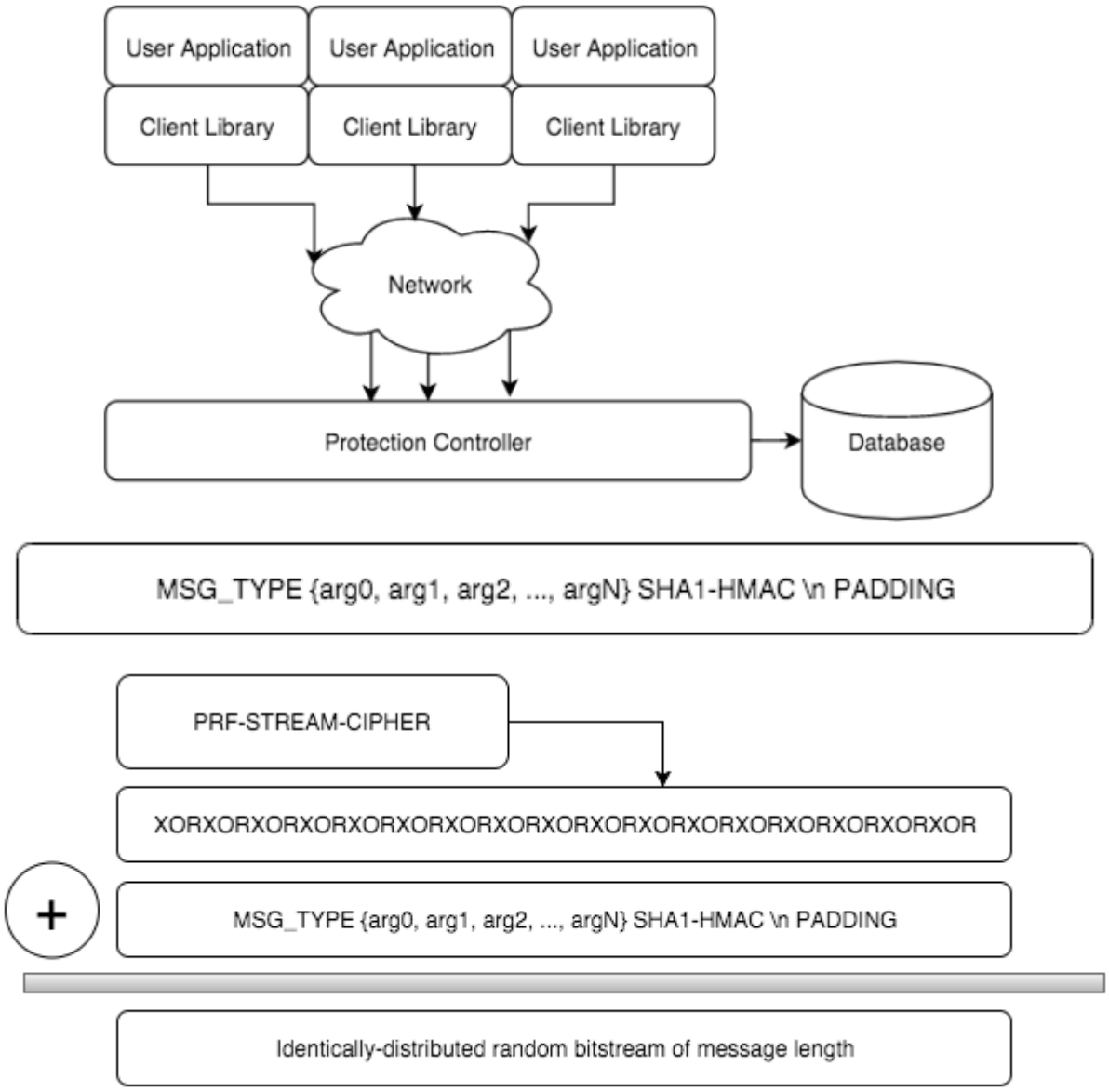
5. SQLite3 emphasizes single-process and single-connection database access while the security daemon should allow simultaneous access.

   The event model provides the illusion of parallel access while interleaved events are processed sequentially. Each event is executed atomically and yields explicitly to other events by scheduling its continuation to run sometime later. A single, global database connection is shared between all events and is associated with the single event dispatch worker thread. Because events are dispatched in serial, we know that each database query runs atomically while events associated with unrelated connections are interleaved. This preserves necessary happens-before relationships and atomicity.

6. C code is often not portable to other platforms, or at least not in the way that Python is portable to all platforms that correctly implement its virtual machine.

   Often, C code gains the characteristics of the platform on which it is written. Bit widths of data types and system calls are not portable from one platform to another. In Python, all integers are arbitrary precision and the precision of floating point numbers is specified by the standard. System calls are abstracted away to platform-agnostic interfaces. To maximize portability, only a minimum C stub to launch the daemon appropriately for the platform is implemented.

Network Architecture



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.

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.

Database Architecture

Table Name	Columns
subjects	subject, key, super
links	subject1, subject2
filters	subject1, subject2, ticket
rights	subject1, ticket, target, object
objects	localpath, dir

The database architecture consists of five simple tables on which all atomic transactions operate. The tables store information about the subjects, links, filters, rights, and objects. The contents of objects are stored on disk but indexed by the database as recommended in the SQLite3 documentation. Composing links and filters is done by the server program. All items are stored as strings to keep the data portable across implementations.

A few errata exist in the current development version that may cause discontinuities between this implementation and SPM:

1. Objects may be directories. Attempting to read from a directory is an error.
2. Links are implemented as bidirectional connections.
3. Any valid filter suffices on a link to enable the subject-initiated transfer of rights
4. A ticket is a string of the form “T\x” where “x” is a right.

The database is accessed in-process and backed by a file store. The database supports recording and rollback of past transactions and an overlay logger of transactions is easily implemented. The database provides data storage and low-level structure only. No logic is implemented here.

Software Architecture

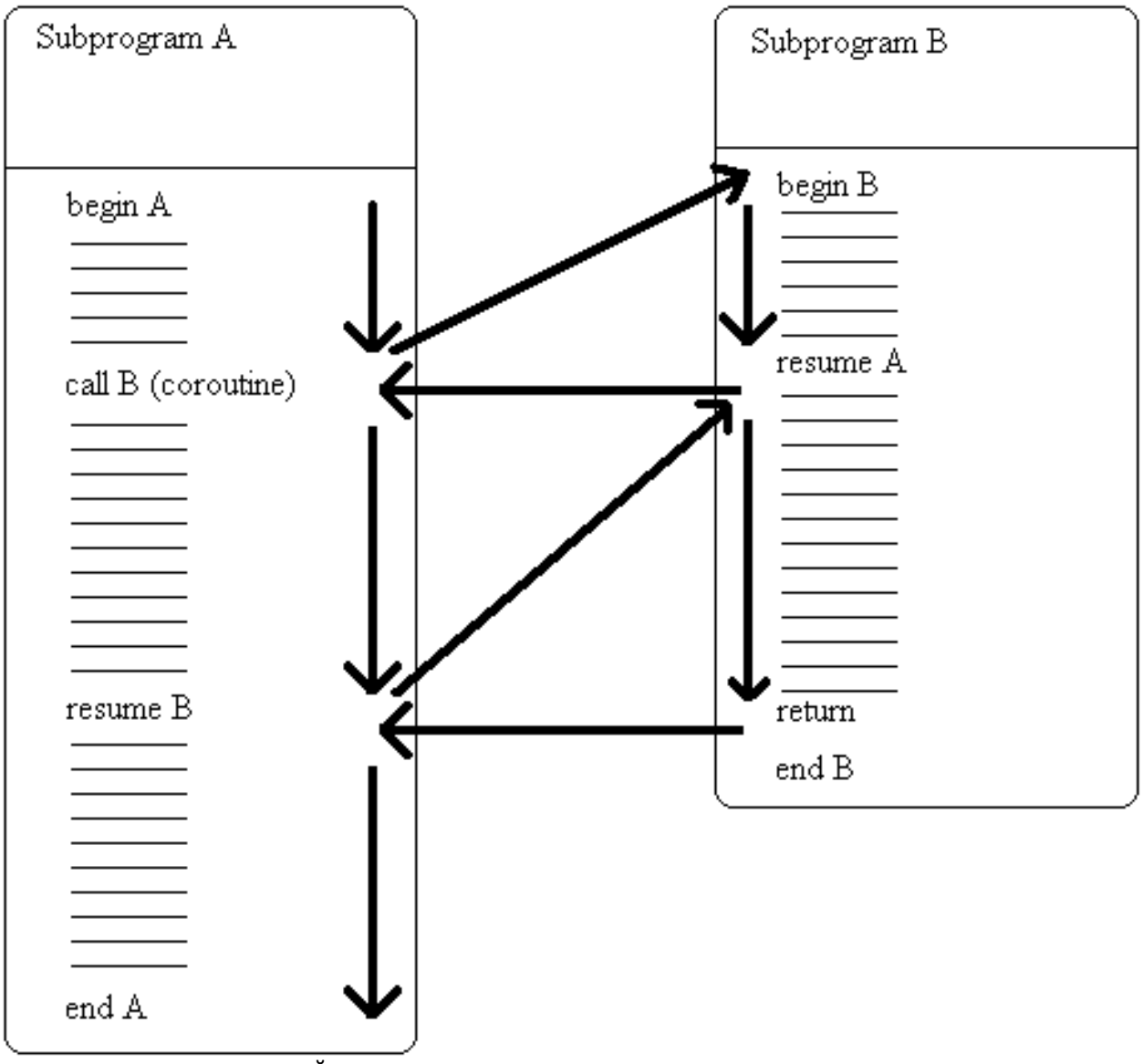


Image Credit: Tevfik AKTUĞLU

The software architecture of this project is based primarily on the following assertion: threads are evil, no matter how they are used. Threading is not an appropriate solution to parallelism because it opens a Pandora's box of new programming errors stemming from violations in happens-before relationships. Race conditions are notoriously hard to diagnose, and few debuggers handle multithreaded applications conveniently or correctly. A better abstraction for a series of instructions to express parallelism is the event, or task, wrapped in a coroutine. This allows fine-grained control of parallelism that is much harder to get wrong. Furthermore, this approach is fully compatible with applications and libraries that are not thread safe because each event is executed atomically.

The previous illustration is freely distributable under the Creative Common's 4.0 share-alike with-attribution license. In the illustration, subprograms A and B were once threads, now represented as continuations. At some point during execution, the continuation yields and schedules itself to continue in the event loop sometime later, allowing other coroutines to continue their execution via a continuation. The core event loop has been implemented and looks like the following:

```
def workerDispatch(self):
  while True:
    try:
      self.dq.popleft()()
    except IndexError:
      sleep(self.idlepoll/1000)
```

The single worker thread blocks on the event loop. The IndexError exception is python-specific for handling non-blocking event queues. When there is work to do, it pops a closure from the event queue and executes the closure in-place. Some of these closures have been implemented. For example, the event (continuation) for accepting an incoming client connection without spawning a new thread:

```
@staticmethod
def acceptClient(dq,socket):
  log_this_func()
  addr = socket.getpeername()
  print("Accepted connection from %s:%i" % addr)
  scope = ClientData(socket)
  dq.append(lambda: Events.readUntilMessageEnd(dq,scope,
      lambda: Events.checkHelloAndReply(dq,scope)))
```

In this continuation, the event accepts a client and initializes local client data structures. When the client has been accepted, instead of calling functions to perform the initial handshake, we schedule a continuation to read from the socket and later perform the handshake with the data read from the socket. When the acceptClient procedure returns, the oldest scheduled coroutine that has yet to run will be executed, and some time later, these continuations will be reached and executed in the order in which the procedures were scheduled.

The continuation is the fundamental building block of my program's control flow, while the closure is the building block of these continuations. Most programs are not written in this manner, but this flavor of event-driven programming allows N to N client to thread mapping, increasing throughput at the cost of some additional latency. The daemon will thus be suited to handling large numbers of simultaneous connections with minimal additional overhead for each additional client. Note that each client does not require its own thread stack space.

Source Tree

The present structure of the source tree is as follows.

LICENSE
: Permissive licensing information releases the work under the MIT license, allowing both commercial and non-commercial use for any purpose while disclaiming all warrantees and fitness of the software for any particular purpose.

README
: Brief overview, project summary, and documentation for the project.

TestClient.py
: Test script exercises the client code for interacting with the server parts of the shared library. It performs simple smoke testing of client functionality.

TestServer.py
: Test script exercises the server code of the shared library. It performs simple smoke testing. Briefly, it initializes a simple server configuration and responds to client requests with some reasonable defaults.

SPM/Client
: High-level client object exposes core functionality through its methods and reports server-generated errors to client code of the library. Users of the client are not required to have any knowledge of the underlying protocol.

SPM/Server
: High-level server object exposes server functionality and configuration. While not strictly necessary, it is recommended to use the Server class to instantiate the daemon to ensure that reasonable and secure defaults are selected.

SPM/Database
: High-level database interface wraps complex transactions into more simple data manipulation requests including, but not limited to, the creation and deletion of users, rules, links, filters, and objects. Throws DatabaseError for illegal operations.

SPM/Events
: Coroutine and continuation implementations. All closures scheduled within the event loop are defined here. An event-driven model based on continuations was chosen to maximize throughput over the traditional low-latency socket server.

SPM/Messages
: Message strategy implementations handle parsing of messages and enforces the appropriate authentication requirements for messages at each access level (before and after authentication). Throws BadMessageError for malformed, illegal, or damaged messages from both trusted and untrusted sources.

SPM/Stream
: Implementation of the stream cipher and HMAC integrity checking.

SPM/Misc
: Small classes and structures representing subjects, rights, and tickets that contain little business logic.

Bibliography

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.

McGrew, D. "AES-GCM and AES-CCM Authenticated Encryption in Secure RTP (SRTP)." *IETF Tools*. The Internet Society, 26 Jan. 2011. Web. 26 Feb. 2016.

Needham, Roger M., and David J. Wheeler. "Tea Extensions." (n.d.): n. pag. Oct. 1997. Web. 26 Feb. 2016.

"Python3 Documentation." *Python.org*. Python Software Foundation, n.d. Web. 26 Feb. 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.