N3533: Standard secure networking

Document #: N3533
Date: 2025-04-21
Project: Programming Language C
Reply-to: Niall Douglas
<s_sourceforge@nedprod.com>

In 2021, after WG21 decided to not proceed with standardising the then Networking TS into C++, I was asked by senior leadership to bring to WG21 an alternative standard networking design with these design goals:

1. There must be a hard ABI boundary across which implementation detail cannot leak. To be specific, if an implementation uses Microsoft SChannel or OpenSSL then absolutely no details of that choice must permeate the ABI boundary.

   Rationale: Standard library maintainers are not in the business of specifying cryptography libraries, and it's a hole nobody rational wants to dig.

2. No imposition must be taken on the end user choice of asynchronous i/o model i.e. if the user wants to use libuv or any other third party async framework, this API is to enforce no requirements on that choice.

   Rationale: A majority of 'simple' use cases for networking just need to operate one or a few sockets, and don't need a full fat async framework or one which can pump millions of concurrent connections per kernel thread. Fully blocking i/o, or non-blocking multiplexed i/o with `poll()` is all they need and having to master a complex async i/o framework just to pump a single socket is not a positive end user experience.

3. Whole system zero copy i/o, ultra low latency userspace TCP/IP stacks, NIC or kernel accelerated TLS and other 'fancy networking tech' ought to be easily exposable by the standard API without leaking any implementation details.

   Rationale: It is frustrating when networking implementations assume that the only networking possible is implemented by your host OS kernel, but you have a fancy Mellanox card capable of so much more. This leads to hacks such as runtime binary patching of networking syscalls into redirects. Avoiding the need for this would be valuable.

4. The design should be Freestanding capable, and suit well the kind of networking available on Arduino microcontrollers et al.

   Rationale: On very small computers your networking is typically implemented by a fixed size coprocessor capable of one, four or maybe eight TCP connections. Being able to write and debug your code on desktop, and it work without further effort on a microcontroller, is valuable. Also, as the proposed API design never unbounded allocates memory, it is suitable for embedded systems without a dynamic `malloc()` available.

5. We should accommodate, or at least not get in the way of, implementer's proprietary networking implementation enhancements. As examples: on one major implementation the only networking allowed is a proprietary secure socket running on a proprietary dynamically scaling async framework; on another major implementation there is a proprietary tight integration between their proprietary secure socket implementation, their whole system zero copy i/o framework, and their dynamic concurrency and i/o multiplexing framework.

   Rationale: Leaving freedom for platforms to innovate leaves open future standardisation opportunities.

I dutifully designed and implemented a reference implementation, and presented it to WG21 in 2022 as [P2586] *Standard Secure Networking*. It was, unfortunately, not warmly received as it was felt to be too 'low level'.

P2586's API is 99% C compatible, so my question to WG14 is this: *Would the C standard like to standardise a similar API instead?* If the committee decides yes, I will port the C++ reference implementation into a C reference implementation based on `https://en.wikipedia.org/wiki/Mbed_TLS`[1] suitable for as-is inclusion into any C standard library – implementers can drop in a copy of that reference implementation into their standard libraries, or roll their own, up to them.

This porting work would be a fair bit of effort, so if I do this work, I would like to not repeat my experience at WG21 where I invest nine months of my free time only to see the proposal dismissed out of hand by the committee. If WG14 could give strong direction based on the examples of use below, I would appreciate it.

1 Examples of what the API would look like

As the C library does not exist yet, these are necessarily mock ups of a potential API. However, they are a direct port from the examples in P2586 slightly adjusted to be C not C++. They therefore would be representative of the semantics of a real, working, implementation.

To be clear, I minimised the name changes from the examples in P2586 to preserve API naming fidelity. In any actual real C library, I would use more C like naming and design. The semantics shown below would be preserved however.

1.1 Example: Retrieving a web page over HTTPS

This example of use shows how one might retrieve a HTML page from a website via Transport Layer Security (TLS) using this proposal:

```c
static constexpr char test_host[] = "github.com";
static constexpr char get_request[] = "GET / HTTP/1.0\r\nHost: github.com\r\n\r\n";

// Get the default TLS socket source for this platform. There
// can be many TLS socket sources, each with different
// properties, and you can filter and enumerate and examine
// them. This just gets the standard library's suggested default.
```

[1]This is a permissively licensed TLS implementation for embedded systems. For desktop and server, at least Ubuntu and RHEL provide maintained system packages which would be dynamically loaded, this means bug fixes etc are not a problem for the standard library maintainer.

```
//
// Note that while this could be an allocated instance, it does NOT
// mean that the instance returned is dynamically allocated!
// It may be a reference counted singleton, or a static singleton.
// This goes for all the other pointers returned in this
// proposal, they can come from an internal fixed size static
// array. No malloc required!
tls_socket_source_t tls_socket_source =
  tls_socket_source_registry_default_source_create();

// Create a multiplexable connecting socket from the TLS socket
// source. Multiplexable means you can do i/o on more than one
// of them at a time per kernel thread (which on POSIX would mean
// they are non-blocking, on other platforms e.g. Windows it means
// they are OVERLAPPED).
tls_byte_socket_t sock =
  tls_socket_source->multiplexable_connecting_socket(AF_ANY);
{
  // Connect the socket to the test host on port 443, timing out
  // after a five second deadline. This sets the TLS session host
  // from the hostname, and the default is to authenticate the TLS
  // certificate of things we connect to using the local system's
  // certificate store. The host name is therefore needed.
  //
  // This is just the convenience API. If you want to, you can go
  // do each step here separately. In fact, this convenience API
  // is actually implemented 100% using other public APIs.
  int r = sock->connect(test_host, 443, 5000);
  if(r != 0)
  {
    if(r == ETIMEDOUT
      || r == EHOSTUNREACH
      || r == ENETUNREACH)
    {
      printf("\nNOTE: Failed to connect to %s"
             " within five seconds. Error was: %s\n",
             test_host, strerror(r));
      return;
    }
    abort();
  }
}

// The string printed here will be implementation defined. A non-normative
// note will suggest item separation with commas.
printf("\nThe socket which connected to %s"
       " negotiated the cipher %s\n", test_host, sock->algorithms_description());

// Create a buffer sequence from which to write the HTTP/1.0 request to the host.
struct iovec get_request_buffer[1] = { { get_request, sizeof(get_request) - 1 } };

// Write the HTTP/1.0 request to the host.
size_t written = sock->write(get_request_buffer, 1);
TEST_REQUIRE(written == sizeof(get_request) - 1);

// Fetch the front page. The connection will close once all data is sent
// due to that being HTTP 1.0 default semantics, and that causes reads to
// return no bytes read. Note the deadline in case remote doesn't close the
// socket in a timely fashion.
char *buffer = malloc(4096);
size_t offset = 0, buffer_size = 4096;
struct iovec vec[1] = { { buffer, buffer_size } };
for(size_t nread = 0; (nread = sock->read(vec, 1, 3000)) != (size_t) -1;)
{
  offset += nread;
  if(buffer_size - offset < 1024)
  {
    buffer = realloc(buffer, buffer_size + 4096);
    buffer_size += 4096;
  }
  vec[0].iov_base = buffer + offset;
  vec[0].iov_len = buffer_size - offset;
}
printf("\nRead from %s %zu bytes. The first 1024 bytes are:\n\n",
       test_host, offset);
fwrite(buffer, 1, (offset < 1024) ? offset : 1024, stdout);
puts("");

// Gracefully shutdown the TLS connection and close the socket
sock->shutdown_and_close();

// Potentially free the socket source
tls_socket_source->destroy();
```

Why this example explicitly needs a multiplexable socket is worth explaining. Firstly, the difference between a multiplexable and a non-multiplexable handle is that the former can require up to two syscalls per i/o, whereas the latter never requires more than one. Non-multiplexable i/o may be more amenable to Direct Memory Access (DMA) in certain circumstances on some platforms than multiplexable, and the kernel scheduler may make more intelligent scheduling decisions if i/o blocks. We therefore default to non-multiplexable as it is the more efficient choice, and it also has portable semantics across all platforms for all types of i/o handle.

Some implementations may be able to implement the above perfectly well with non-multiplexable sockets, however some may refuse non-zero non-infinite deadline i/o on a non-multiplexable socket (just to be clear here, an implementation may support zero length waits or infinite waits but nothing in between). In the above, we accept the efficiency hit for convenience.

1.2 Example: Multiplexing TLS server connections

One can create listening TLS sockets, and poll for new things happening on them:

```
const char *host;  // set to one of my network cards

// Get me a FIPS 140 compliant source of TLS sockets which uses kernel
// sockets, this call takes bits set and bits masked, so the implementation
// features bitfield is masked for the FIPS 140 bit, and only if that is set
// is the TLS socket source considered.
static constexpr tls_socket_source_implementation_features required_features =
  tls_socket_source_implementation_features_FIPS_140_2
  | tls_socket_source_implementation_features_kernel_sockets;
tls_socket_source_t tls_socket_source =
  tls_socket_source_registry_default_source(required_features,
    required_features)->create();

// Create a listening TLS socket on any IP family.
listening_tls_socket_t serversocket = tls_socket_source
  ->multiplexable_listening_socket(AF_ANY);

// The default is to NOT authenticate the TLS certificates of those connecting
// in with the system's TLS certificate store. If you wanted something
// different, you'd set that now using:
// serversocket->set_authentication_certificates_path()

// Bind the listening TLS socket to port 8989 on the NIC described by host
tls_addr_t addr = tls_socket_source->make_address(host, 8989);
serversocket->bind(addr);
addr->destroy();

// poll() works on three arrays. We deliberately have separate poll_what
// storage so we can avoid having to reset topoll's contents per poll()
tls_byte_socket_t *pollable_handles = calloc(1024, sizeof(tls_byte_socket_t));
poll_what *topoll = calloc(1024, sizeof(poll_what));
poll_what *whatchanged = calloc(1024, sizeof(poll_what));

// We will want to know about new inbound connections to our listening
// socket
pollable_handles[0] = serversocket;
topoll[0] = poll_what_is_readable;
whatchanged[0] = poll_what_none;

// Connected socket state
struct connected_socket {
  // The connected socket
  tls_socket_handle_t socket;

  // The endpoint from which it connected
  tls_addr_t endpoint;

  // The size of registered buffer actually allocated (we request system
  // page size, the DMA controller may return that or more)
  size_t rbuf_storage_length;
  size_t wbuf_storage_length;

  // These are ptrs to memory potentially registered with the NIC's
  // DMA controller and therefore i/o using them would be true whole system
  // zero copy
  tls_socket_handle_registered_buffer_t rbuf_storage, wbuf_storage;

  // These slices of byte and const byte index into buffer storage and get
  // updated by each i/o operation to describe what was done
  struct { size_t len; struct iovec *iov; } rbufs;
  struct { size_t len; struct iovec *iov; } wbufs;
};
struct connected_socket *connected_sockets = calloc(1024, sizeof(struct connected_socket));
size_t connected_sockets_len = 1;
```

```

// Begin the processing loop
for(;;) {
  // We need to clear whatchanged before use, as it accumulates bits set.
  FILL(whatchanged, 1024, poll_what_none);

  // Wait until the deadline for something to change
  size_t handles_changed
    = tls_socket_source->poll(whatchanged, pollable_handles, topoll,
      connected_sockets_len, -1 /* infinite timeout */);

  // Loop the polled handles looking for events changed.
  for(size_t handleidx = 0;
    handles_changed > 0 && handleidx < 1024;
    handleidx++) {
    if(whatchanged[handleidx] == poll_what_none) {
      continue;
    }
    if(0 == handleidx) {
      // This is the listening socket, and there is a new connection to be
      // read, as listening socket defines its read operation to be
      // connected sockets and the endpoint they connected from.
      struct pair { tls_socket_handle_t sock; ipaddr_t endpoint; } s;
      struct iovec[1] = { { &s, sizeof(s) } };
      serversocket->read(iovec, 1);
      connected_sockets[connected_sockets_len].socket = s.sock;
      connected_sockets[connected_sockets_len].endpoint = s.endpoint;
      connected_sockets[connected_sockets_len].rbuf_storage_length = getpagesize();
      connected_sockets[connected_sockets_len].wbuf_storage_length = getpagesize();
      connected_sockets[connected_sockets_len].rbuf_storage =
        socket->allocate_registered_read_buffer(
          &connected_sockets[connected_sockets_len].rbuf_storage_length);
      connected_sockets[connected_sockets_len].wbuf_storage =
        socket->allocate_registered_write_buffer(
          &connected_sockets[connected_sockets_len].wbuf_storage_length);
      connected_sockets[connected_sockets_len].rbufs =
        connected_sockets[connected_sockets_len].rbuf_storage->iovecs();
      connected_sockets[connected_sockets_len].wbufs =
        connected_sockets[connected_sockets_len].wbuf_storage->iovecs();

      // Watch this connected socket going forth for data to read or failure
      pollable_handles[connected_sockets_len] = s.sock;
      topoll[connected_sockets_len] = poll_what_is_readable | poll_what_is_errored;
      connected_sockets_len++;
      handles_changed--;
      continue;
    }
    // There has been an event on this socket
    auto *sock = &connected_sockets[handleidx - 1];
    handles_changed--;
    if(whatchanged[handleidx] & poll_what_is_errored) {
      // Force it closed and ignore it from polling going forth
      sock->socket->close();
      sock->rbuf_storage->destroy();
      sock->wbuf_storage->destroy();
      pollable_handles[handleidx] = nullptr;
    }
    if(whatchanged[handleidx] & poll_what_is_readable) {
      // Set up the buffer to fill. Note the scatter buffer list
      // based API.
      size_t bytesread = sock->socket->read(sock->rbufs.iov, 1);
      // Do something with the buffer
      // Reset the buffer
      sock->rbufs = sock->rbuf_storage->iovecs();
    }
    if(whatchanged[handleidx] & poll_what_is_writable) {
      // If this was set in topoll, it means write buffers
      // were full at some point, and we are now being told
      // there is space to write some more
      if(sock->wbufs.iov[0].iov_len > 0) {
        size_t byteswritten = sock->socket->write(sock->wbufs.iov, 1);
        sock->wbufs.iov_base += byteswritten;
        sock->wbufs.iov_len -= byteswritten;
      }
      if(sock->wbufs.iov[0].iov_len == 0) {
        // Nothing more to write, so no longer poll for this
        topoll[handleidx] & =~poll_what_is_writable;
        // Reset the buffer
        sock->wbufs = sock->wbuf_storage->iovecs();
      }
    }

    // Process buffers read and written for this socket ...
  }
}
```

poll() has by definition linear scaling to sockets being polled, and as a result most OS kernels refuse to accept more than a few hundred inputs per poll(). Despite this, this simple method of multiplexing i/o on a single kernel thread will get you surprisingly far in most real world use cases for i/o multiplexing – only when you operate thousands of concurrent connections per kernel thread do scalability issues emerge, and then becomes important alternative techniques such as epoll or io_uring for Linux, Grand Central Dispatch for Apple Mac OS, or IOCP and RIO for Microsoft Windows. I would argue that WG14 does not need to standardise anything beyond poll() at the present time. Absolutely leave open potential for future standardisation in that area, but let's leave that off for now, and standard library implementators can of course add additional APIs to support their proprietary i/o reactor if they wish.

1.3 Example: Wrapping a third party socket implementation with TLS

```
// I want a TLS socket source which supports wrapping externally
// supplied byte_socket_handle instances
static constexpr tls_socket_source_implementation_features required_features =
  tls_socket_source_implementation_features_supports_wrap;
tls_socket_source_t tls_socket_source =
  tls_socket_source_registry_default_source(required_features,
    required_features)->create();
byte_socket_source_t byte_socket_source =
  byte_socket_source_registry_default_source_create();

```

```
// Create raw kernel sockets.
byte_socket_t rawsocket
  = byte_socket_source->connecting_socket(AF_ANY);

listening_byte_socket_t rawlsocket
  = byte_socket_source->listening_socket(AF_ANY);

// Attempt to wrap the raw socket with TLS. Note the "attempt" part,
// just because a TLS implementation supports wrapping doesn't mean
// that it will wrap this specific raw socket, it may error out.
//
// Note also that no ownership of the raw socket is taken - you must
// NOT move it in memory until the TLS wrapped socket is done with it.
tls_socket_handle_t securesocket
  = tls_socket_source->wrap(rawsocket);
listening_tls_socket_handle_t serversocket
  = tls_socket_source->wrap(rawlsocket);
```

To support non-standard socket implementations, TLS socket sources may be able to wrap a third party supplied socket implementation – any implementation of `byte_socket_handle` or `listening_byte_socket_handle` which meets the API guarantees is suitable, and this can include unit test suite fake sockets, pipes, or any arbitrary transport which quacks like a socket. There isn't much more to say here, other than that this facility would exist in the proposed standardisation.

2 Why not IETF TAPS instead?

Since when [P2586] was rejected by WG21, [P3482] *Design for C++ networking based on IETF TAPS* has been proposed for C++. This would add to the C++ standard library an implementation of `https://datatracker.ietf.org/doc/rfc9622/`, which is the IETF TAPS standard.

There are similarities between this proposal and IETF TAPS:

1. Both have a queryable registry of transport implementations which manufacture instances.
2. Both are compatible with Apple's Network Framework, though for different reasons.

There are many differences however:

1. This proposal supports only reliable stream based transports. TAPS supports packet based, datagram based, and many others.
2. This proposal supports only TLS as the security layer. TAPS theoretically supports security layers not yet designed.
3. This proposal is designed to work very well on embedded systems where networking is provided by a coprocessor, and where dynamic `malloc()` may not be available.
4. This proposal does one thing specifically, whereas TAPS is absolutely enormous and covers every conceivable type of networking connection – packet-orientated, message-orientated etc – and includes potential future types of networking connection for which the application was not intentionally written.

5. This proposal supports blocking and non-blocking i/o only, whereas TAPS requires an asynchronous i/o implementation.
6. This proposal rather hand waves away how to handle TLS certificates as standardising that portably is very hard (we provide a 'name' setting with platform-specific meaning, and we allow enable/disable of 'whatever the system defines as certificate validity'. Everything else is left to platform-specific APIs). TAPS has a full fat attempt to standardise trust verification and identity challenge across platforms, and speaking personally I'm not 100% sure what they propose is implementable in some use cases.

In short, I think TAPS is great as an ecosystem supplied library. I think it far too large for the C standard library. I would point out that this proposal would be one of several potential implementation backends for a TAPS implementation, but you would need a lot more code to achieve a compliant TAPS.

3 Conclusion

Being able to connect to a web server and retrieve a HTTPS page has become one of the fundamental things software needs to do. The technologies involved – sockets and TLS – are mature by this point.

The question I put to WG14 is this: should the standard C library provide a standard socket and TLS implementation?

The two main arguments against I think are:

1. **The standard C library should not standardise something that the ecosystem ought to provide.**

   This is fair, however there is the portability argument, and when US$2 microcontrollers can do TLS based secure networking, I think there is a fair case that standardising a portable API has become time.

2. **Is not secure networking still evolving quickly i.e. HTTP/3?**

   Every HTTP/3 server will continue to provide HTTP/2 same as they continue to provide HTTP/1.1. There is also a large ecosystem of non-HTTP TLS secured socket based services out there which will stay using TLS forever, not least because HTTP/3 uses UDP rather than TCP as its underlying transport e.g. SSH is very likely to remain TLS based for decades to come for the simple reason that it's plenty good enough for that use case.

   This proposal enables standard HTTPS page fetches, but you still have to implement all the HTTP stuff yourself. This proposal would only standardise the secure networking part of things, and I don't see that technology changing much any time soon.

4 References

[P2586] Douglas, Niall
WG21 P2586: *Standard Secure Networking*
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2586r0.pdf

[P3482] Rogers, Thomas; Kühl Dietmar
WG21 P3482: *Design for C++ networking based on IETF TAPS*
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p3482r0.html