Reliable User Space TLS tracing with eBPF

Tracing Summit 2023

Dom Del Nano

Pixie: eBPF-based Observability for K8s

- Observability tool that provides full fidelity protocol traces between your microservices through auto instrumentation.
- Supports many popular protocols (grpc, HTTP, mysql, etc) and can trace TLS encrypted connections.
- TLS is widely adopted in today's environments. Being unable to trace these connections creates substantial blind spots

Table

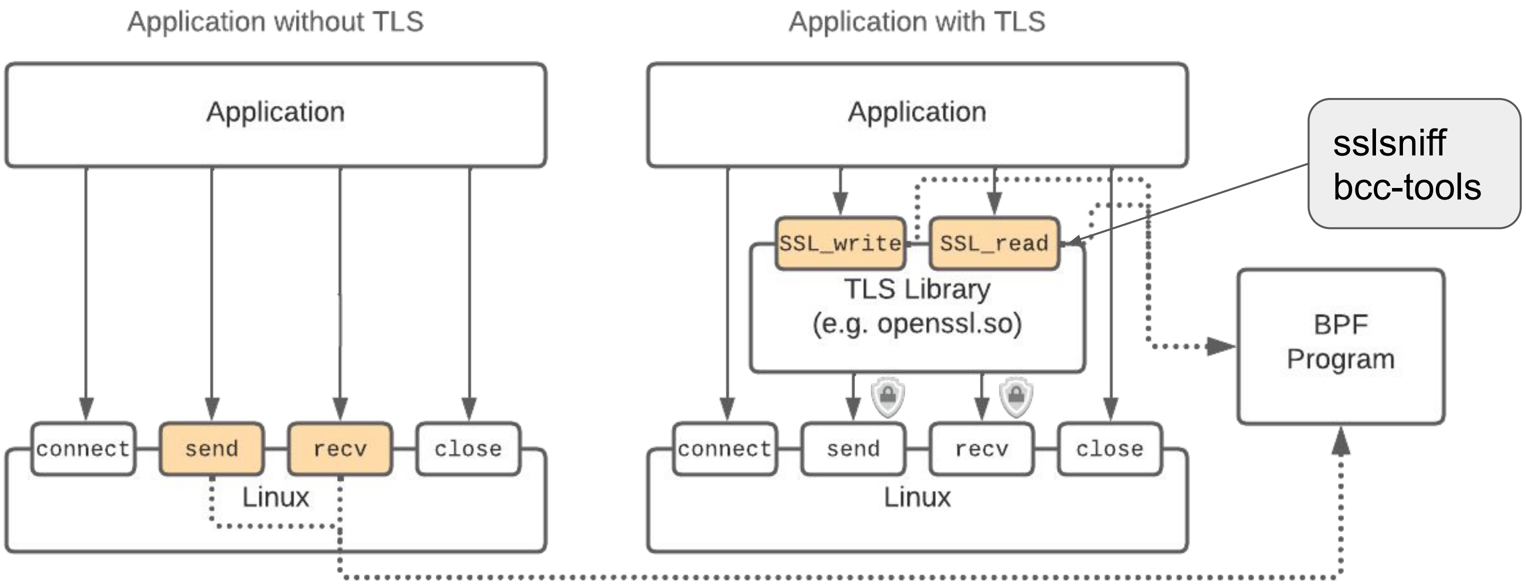
TIME_	SOURCE	DESTINA...	LATENCY	REQ_PATH	REQ_...	REQ_BODY	RESP...	RESP_BODY
5/11/2022, 2...	sock-shop/o...	sock-shop/u...	1.5 ms	/addresses/57a98d98e4b00679b4a830b0	GET		200	{ street: Whitelees Road, number: 246, country: United Kingdom, city: G...
5/11/2022, 2...	sock-shop/o...	sock-shop/u...	850.1 µs	/cards/57a98d98e4b00679b4a830b1	GET		200	{ longNum: 5544154011345918, expires: 08/19, ccv: 958, id: 57a98d98e4b0...
5/11/2022, 2...	sock-shop/o...	sock-shop/u...	5.3 ms	/customers/57a98d98e4b00679b4a830b2	GET		200	{ firstName: User, lastName: Name, username: user, id: 57a98d98e4b00679...
5/11/2022, 2...	sock-shop/o...	sock-shop/c...	5.4 ms	/carts/57a98d98e4b00679b4a830b2/items	GET		200	[{ id: 627b7b1cc21cd70006538526, itemId: 819e1fbf-8b7e-4f6d-811f-69353...
5/11/2022, 2...	sock-shop/o...	sock-shop/p...	591.9 µs	/paymentAuth	POST	{ address:...	200	{ authorised: true, message: Payment authorised }
5/11/2022, 2...	sock-shop/o...	sock-shop/s...	2 ms	/shipping	POST	{ id: 75ea...	201	{ id: 75ead273-5a18-4e69-9f52-deb56c9b9d0f, name: 57a98d98e4b00679b4a83...

Agenda

- Overview of TLS tracing and why handling User space is unavoidable
- Deep dive on Pixie's initial form of TLS tracing and its challenges
- Discuss the latest tracing and how it handles the complex challenges more elegantly
- Future work

TLS Tracing Introduction

- Encryption often occurs within a user space library (OpenSSL, BoringSSL)
- Tracing user space is unavoidable for tracing TLS

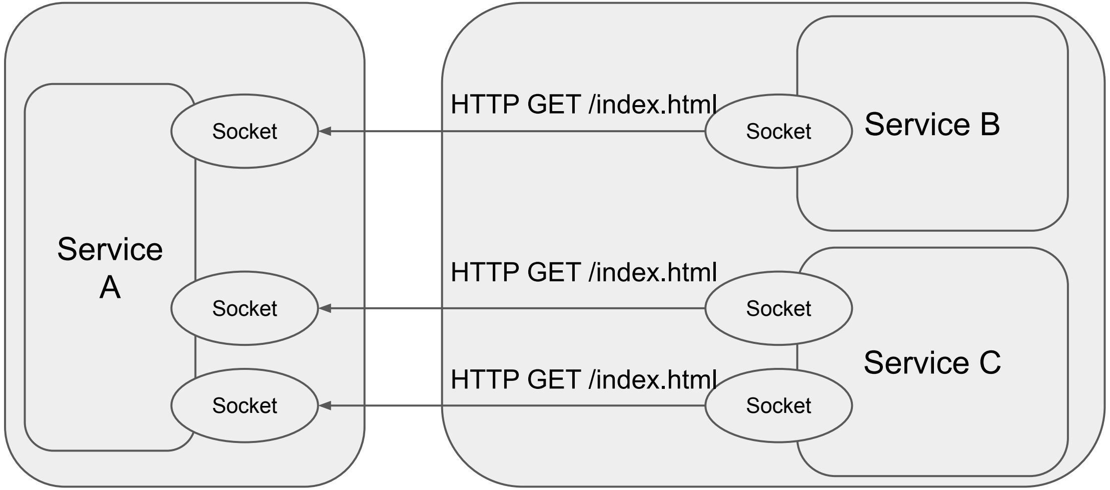


TLS Tracing Production Use Cases

- In reality, tracing production systems comes with more challenges:
  - Different types of linking (dynamic, static)
  - Many popular libraries (OpenSSL, BoringSSL, LibreSSL, GnuTLS, etc)
  - Different ways a given library can be interfaced with
- These use cases require more than the plaintext data
  - Environments today have many microservices and their tracing data must contain additional metadata to make it useful.

Challenges Tracing Encrypted Messages

- Accessing the protocol data is just part of the story
  - The network traffic must be attributed to a particular connection to make the data usable.
  - The connection must be identified so the socket file descriptor must be accessible



Challenges Tracing Encrypted Messages

- Plaintext protocol tracing has easy socket fd access from syscall parameters
- Socket file descriptor is not part of the OpenSSL API and must be accessed through another mechanism

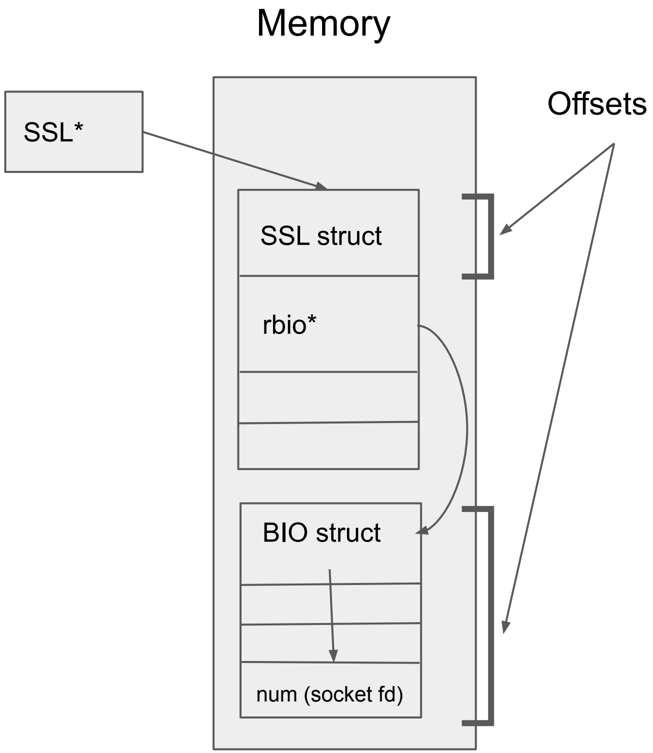
```
ssize_t send(int sockfd, const void buf, size_t len, int flags);


int SSL_write(SSL *ssl, const void *plaintext, int num);
typedef struct ssl_st SSL;
typedef struct bio_st BIO;

struct ssl_st {
    BIO *rbio;
    BIO *wbio;
    [ … ]
}
struct bio_st {
    int num;             <-----   Stores the socket file descriptor
}
```

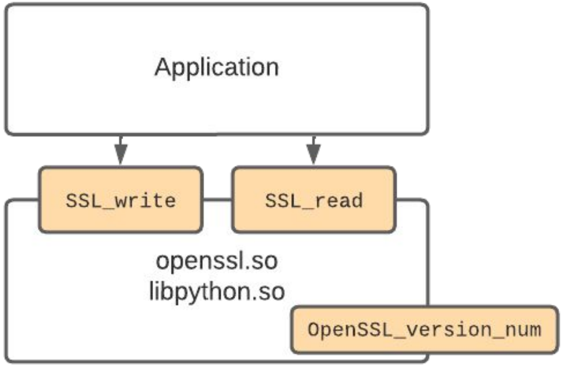
Challenges Tracing Encrypted Messages

- Initial tracing used memory offsets to access the socket fd
    - Assume stable offsets for a given OpenSSL version.
- This created another challenge – reliably detecting the OpenSSL version.
- Version detection initially relied on *OpenSSL_version_num* function but became more challenging as more libraries and linking options were in scope



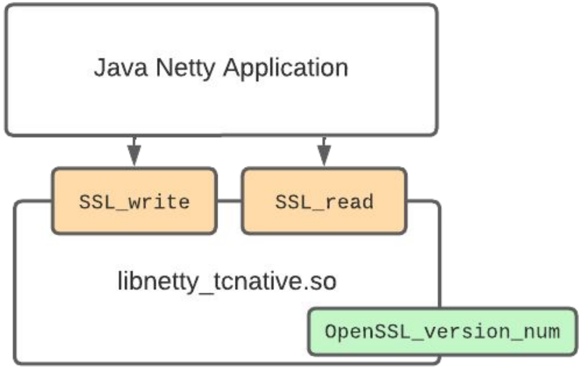
OpenSSL Version Detection

OpenSSL, Python



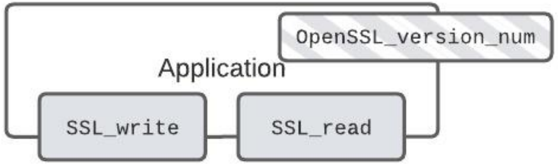
✅dlsym(handle, "OpenSSL_version_num")

Netty-tcnative (BoringSSL variant)



❌dlsym(handle, "OpenSSL_version_num")
✅RawSymbolToFptr<T>("OpenSSL_version_num")

Statically linked OpenSSL / BoringSSL

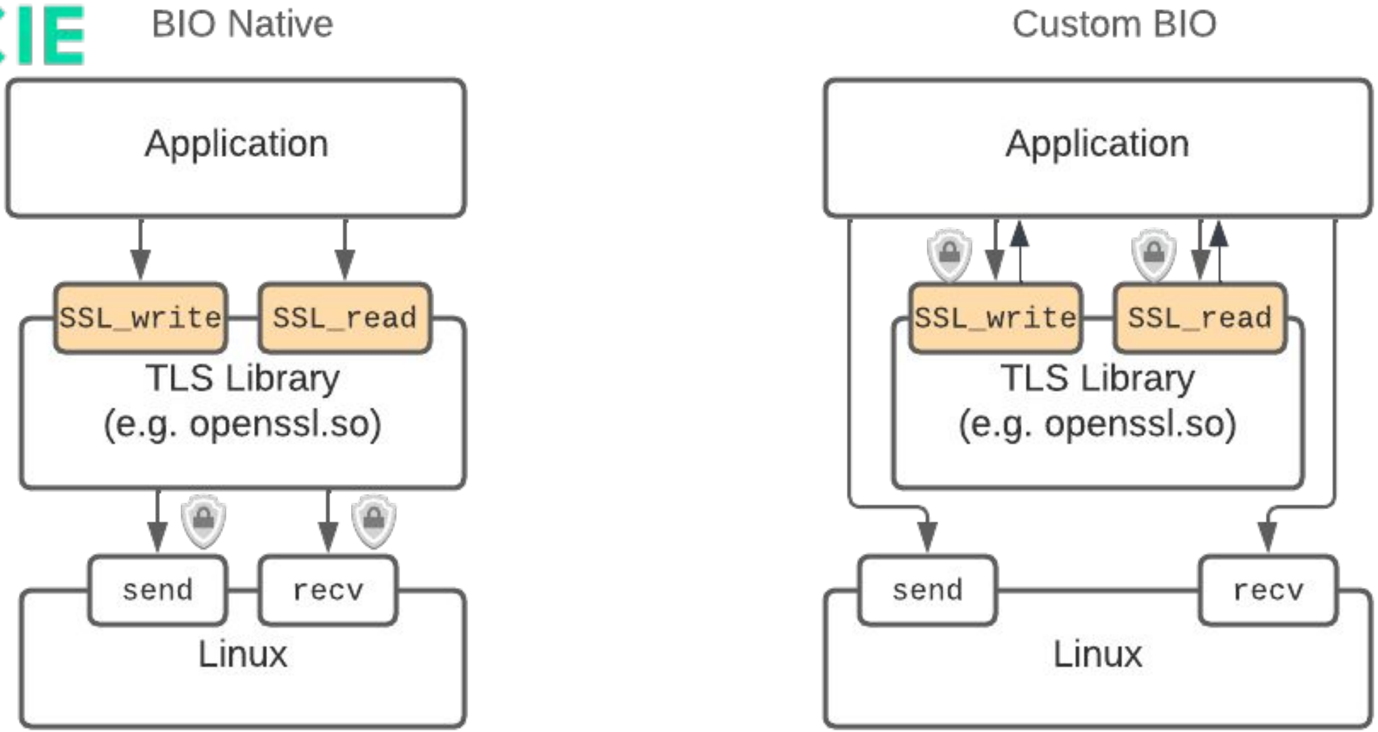


❌dlsym(handle, "OpenSSL_version_num")
❌RawSymbolToFptr<T>("OpenSSL_version_num")

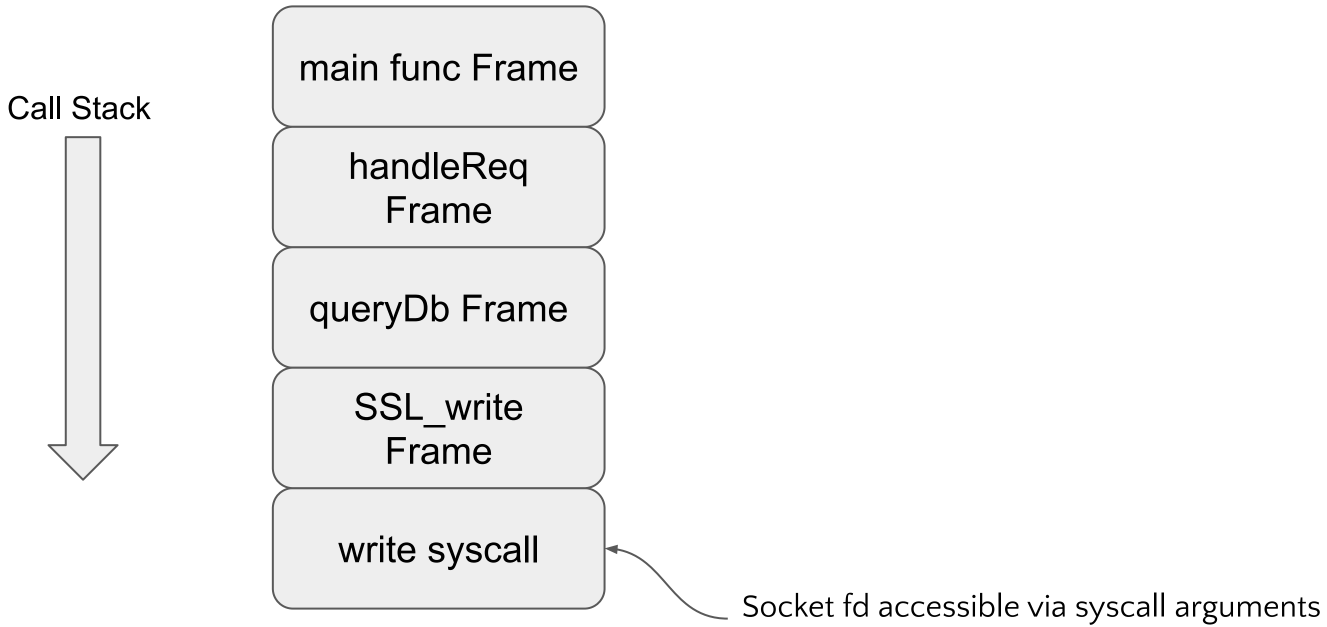
Redesigned TLS Tracing

- Standardizing socket fd access appeared too challenging once BoringSSL (static linking) was considered
- Was this the right problem to solve? Relying on user space offsets with no stability guarantees caused more difficult challenges.
- OpenSSL and compatible libraries can be classified in the following ways:
    - BIO Native
        - OpenSSL manages the IO to the underlying socket. Socket is expected to be populated on SSL struct
        - Examples: Nginx, Python
    - Custom BIO
        - OpenSSL is used for encryption exclusively. Application handles IO itself and usually done async (with an event loop)
        - Examples: NodeJS, Envoy

BIO Native vs Custom BIO



BIO Native Deep Dive



Redesigned TLS Tracing

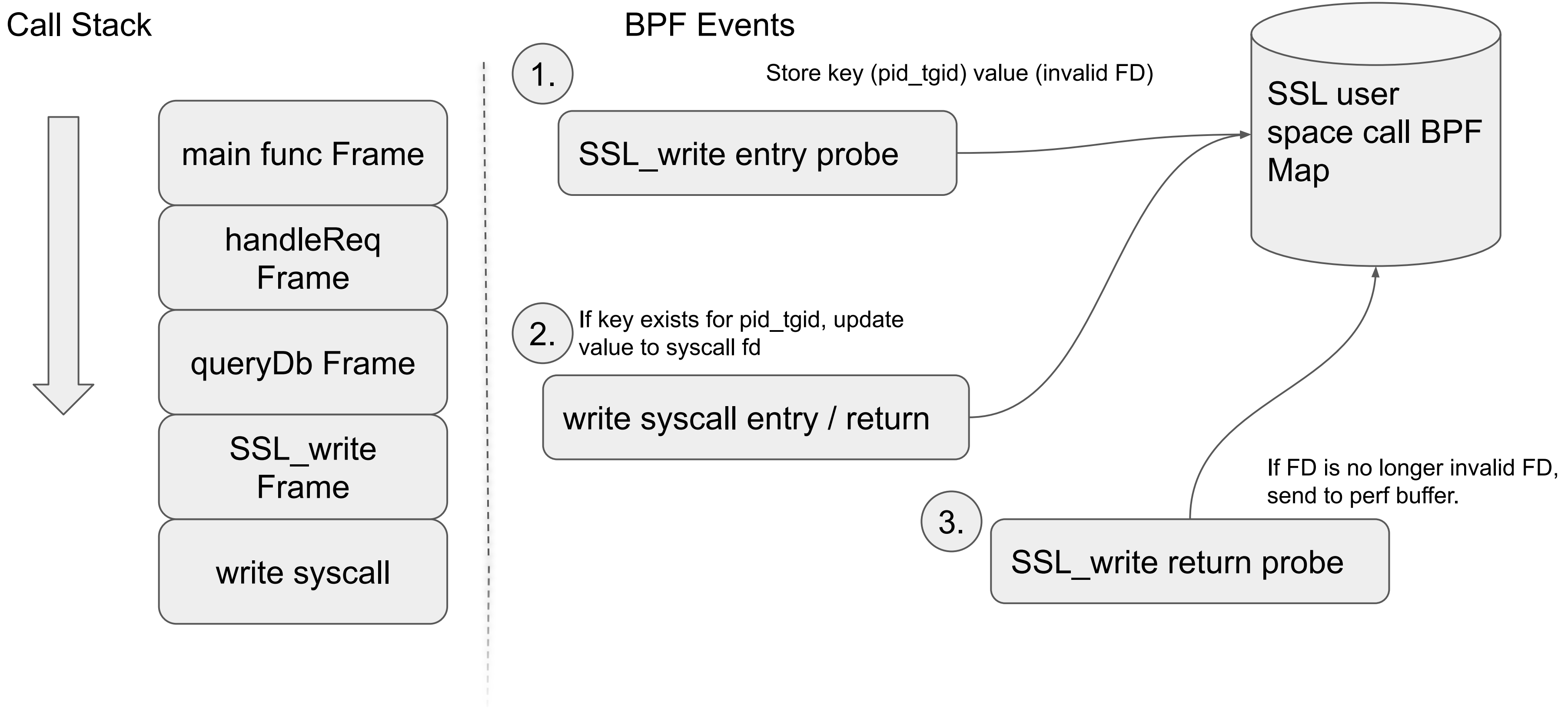
- For BIO native applications, assume that socket syscalls will occur while SSL_write/SSL_read are on the stack*
- This provides an opportunity to pass the socket fd from the nested syscall to user space on the uretprobe.
  - This would have the potential to remove all reliance on user space offsets and would avoid the ongoing maintenance of the existing tracing.

* OpenSSL does have the ability to perform async operations via custom engines. This allows for hardware offload (Intel QAT) and other advanced features.

Validating call stack assumptions

- This implementation relies on the assumptions about the call stack
- Primary concern was if unrelated io / syscalls (different connections) occurred while the TLS library calls are on the stack
- Developed integrity checking into the TLS tracing implementation
  - If more than one syscall occurs between TLS library calls, the fd must be the same throughout – (i.e. buffered writes)
- This integrity check has identified 5 programs that violate this assumption
  - 99.3% of clusters do not see this condition. Half of which belong to the same end users.
  - 99.9376% of total integrity checks are successful

Redesigned TLS Tracing



TLS Tracing Coverage Review

Application	Library	Linking	Library Interface	Initial Impl.	Traced w/ App. Specific Impl.	Redesigned Impl.
Nginx	OpenSSL v1.1.0	Dynamic	BIO Native	✅	N/A	✅
Nginx	OpenSSL v1.1.1	Dynamic	BIO Native	✅	N/A	✅
Nginx	OpenSSL v3.x	Dynamic	BIO Native	🚧	N/A	✅
Python <= 3.9	OpenSSL v1.1.x	Dynamic	BIO Native	✅	N/A	✅
Python >= 3.10	OpenSSL v3.x	Dynamic	BIO Native	🚧	N/A	✅
Clickhouse*	BoringSSL	Static	BIO Native	❌	N/A	✅
NodeJS	OpenSSL	Static	Custom BIO	❌	✅	❌
Netty Tcnative	BoringSSL	Static	Custom BIO	❌	✅	❌
Envoy	BoringSSL	Static	Custom BIO	❌	❌	❌

Future Work

- Better support Custom BIO use cases
    - Investigate remove implementation specific tracing.
    - Ideally this would provide broad coverage with supporting additional applications (Envoy, Istio, etc).
- Handle statically linked cases where symbols are completely stripped.

Thank You