Thread Local Storage in the Fold Framework

by

Lucie Mermod

Abstract

Fold is a framework to create Rust-based (dynamic) linkers, offering simple tools to design and implement new linkers. This work aims to implement fully featured Thread Local Storage support which was lacking in Fold's first implementation.

Date: December 2025

Contents

1 Motivation

Most of nowadays applications rely on multi-threading to efficiently achieve their goal, e.g. by processing data through streaming pipelines or by dispatching different tasks accross threads. This means that for any loader aiming at being usable in a real environment, it must provide an API adequate for multi-threading.

Luckily, there exists since 2002 a specification describing how ELF should store thread-related data, and how loaders need to handle it [1]. This document concerns many architectures other than `x86_64`, so I will summarize the relevant parts Section 2.1.

2 Background

Before diving into how Fold implements Thread Local Storage (TLS), we first need to take a look at what it is exactly, as well as how Musl's standard library [2] interacts with it. I would like to thank Chao-tic [3], MaskRay [4] and the Android developpers [5] for their blogs which helped a lot in understanding how TLS works.

2.1 Thread Local Storage

Thread Local Storage (TLS in short) is the in-memory data structure holding all the data which is unique to a thread, i.e. the Thread Control Block (TCB) and all static variables denoted with `__thread_local`, whether they are defined in the executable's ELF or in any of the shared objects it depends on.

Each ELF object can contain at most one segment marked with `PT_TLS`, indicating that this segment should be loaded as part of the TLS structure. Such segment is called a TLS module and is characterized, as all segments, with its offset in the file, size in the file, size in the memory, etc. If the size in memory is larger than the size in the file, the remaining bytes should be zeroed out when the module is loaded. This segment is usually split into two sections, named `.tdata` and `.tbss`, analogous to the `.data` and `.bss` sections.

The TLS specification [1] describes two memory layouts to hold the TLS, but we will focus on the second variant as it is the one used in x86_64. The figure below gives an overall view of the structure:

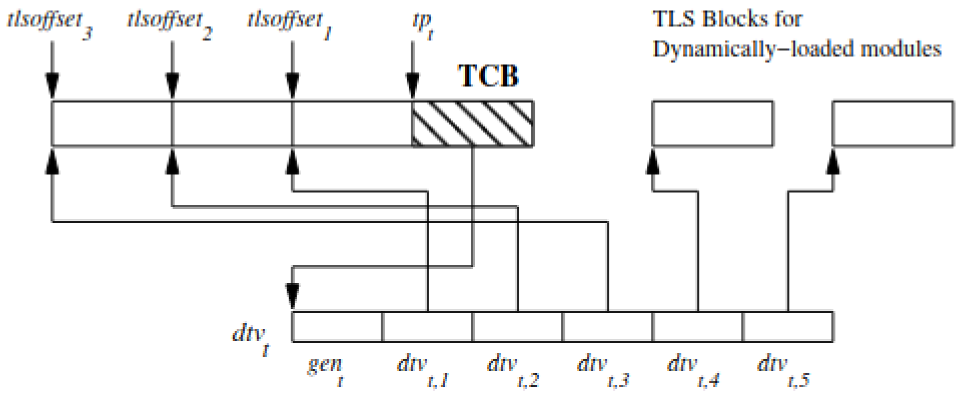

Figure 1: TLS in-memory layout in x86_64

There are four main components:

- Thread control block (TCB): holds thread-specific information such as the Thread Identifier (TID), stack canary, etc. It also contains a pointer to the DTV (below).
- Static modules (at tlsoffset$_x$): TLS modules which are loaded at load-time or during thread creation.
- Dynamic modules: TLS modules which are loaded only when the thread attempts to access memory within them.
- Dynamic Thread Vector (DTV): stores a pointer to each loaded modules, as well as a generation number used when library are dynamically loaded at run-time (using `dlopen`).

When creating the TLS, the loader must identify all the TLS modules across the executable and its dependencies, and assign a Module ID to each of them. IDs starts at 1, which is reserved for the executable's TLS module, if any. Other modules can technically be assigned with any IDs, although for practical reasons the loader will usually simply count from 2.

There are two reasons for the existence of dynamic modules. First, it makes the implementation of `dlopen()` and related functions much easier and safer as they are allowed to place the TLS modules of the new objects anywhere in the address space, and not specifically before the existing modules (this could result in clashing with another existing memory mapping). Also, some modules could be ignore by some threads and loading them would be a waste of time and memory, hence the posssibilty to defer this allocation when the module is actuall accessed (see Code snippet 1 in General Dynamic accesses below).

When a thread wants to access a variable from a TLS module, it can proceed with different methods, by decreasing genericity:

- **General Dynamic**: Using different relocations, the loader will allocate a struct in the GOT to store the Module ID and Offset of variable within the module, which the runtime will then pass to the `__tls_get_addr(uintptr_t[2])` function which will return the actual address of the variable. This function should be provided by the loader itself. A naive implementation would look like this:

```c
void* __tls_get_addr(uintptr_t[2] req) {
  // Allocate the module with id req[0]
  if (tcb->dtv[req[0]] == NULL) {
    allocate_tls_module(req[0]);
  }

  return tcb->dtv[req[0]] + req[1];
}
```

Code snippet 1: simple `__tls_get_addr` implementation

- **Local Dynamic**: If the code accessing the variable is from the same object as the TLS module requested, then it may call the `__tls_get_addr()` function and pass an offset of `0` to get the address of the TLS module, and the compiler can add the offset itself since it would be known at compile time. This method is only interseting if the program accesses multiple thread-local variables, as it reduces to one the number of calls to `__tls_get_addr()`.

- If the module is statically loaded, then the program can access the variable by offsetting the thread pointer. To do so, it can either:
  - **Initial-Exec**: Request the loader to generate a GOT entry to store the offset

- **Local-Exec**: Produce a relocation asking the loader to directly write the offset into the code.

Note that the specification states that Local-Exec cannot be used from a shared object, but I did not understand why this restriction exists.

It is unclear who, between the programmer, compiler or loader, decides which module should be statically loaded. While a `DF_STATIC_TLS` flag exists in the Dynamic Table, I found that it is not produced by compilers and cannot be the sole decider whether the modules should be static as this decision also depends on how other objects access it (which is obviously not known when the compiler generates the shared object). From my understanding, any module may be static except for those accessed through the `R_X86_64_TPOFF32` and `R_X86_64_TPOFF64` (used in Initial-Exec and Local-Exec) which have to be static. Those considerations taken into account, the loader is free to choose whether to allocate dynamically or not the other modules.

2.2 Musl's implementation

Knowing the fundamentals of Thread Local Storage, let's now take a look at the implementation details of Musl.

2.2.1 Module allocation

The first observation to be made is that Musl provides itself an implementation for `__tls_get_addr` (Code snippet 2) which simply indexes the `dtv` array, meaning that all TLS modules are statically allocated. This greatly simplifies to process of allocating the TLS modules, but also increases the overhead of loading the ELF and creating new threads. See also Section 4.1.

```
void *__tls_get_addr(tls_mod_off_t *v)
{
  pthread_t self = __pthread_self();
  return (void *)(self->dtv[v[0]] + v[1]);
}
```

Code snippet 2: Musl's implementation of `__tls_get_addr`

2.2.2 Thread Control Block

On top of the Thread Control Block specified in the ABI (todo: ref wanted), Musl augments it with quite a lot of fields, as shown in Code snippet 9 of Appendix A. Luckily, a lot of these fields are handled by Musl's runtime and should be initialized to `0`; and the function `__init_tp` (Code snippet 3) shows us which fields should be initialized and how.

```
int __init_tp(void *p)
{
  pthread_t td = p;
  td->self = td;
  int r = __set_thread_area(TP_ADJ(p));
  if (r < 0) return -1;
  if (!r) libc.can_do_threads = 1;
  td->detach_state = DT_JOINABLE;
  td->tid = __syscall(SYS_set_tid_address, &__thread_list_lock);
  td->locale = &libc.global_locale;
  td->robust_list.head = &td->robust_list.head;
  td->sysinfo = __sysinfo;
  td->next = td->prev = td;
  return 0;
}
```

Code snippet 3: `__init_tp`

2.2.3 The `libc` object

When inspecting the `__init_tp` function (Code snippet 3), we encounter a rather peculiar object called `libc` (Code snippet 4). It is a global variable in the Musl library that stores general informations about the runtime. Here, we will focus on the thread-related fields which are for the most part quite simple.

- `can_do_threads`: Whether the TLS was correctly initialized. Needs to be set to `1` in order for the `thrd_create` function to succeed.
- `threaded`: Handled by Musl upon first call of `thrd_create`, indicates whether the library has initialized its thread-related components.
- `threads_minus_1`: Number of threads running, minus 1.
- `tls_head`: A linked list of `tls_module` (Code snippet 5) objects. This list is used to create the TLS region when an new thread is insantiated. It must be ordered according to the Module IDs.
- `tls_size`: Total size in bytes of the TLS region, ergo size of the TCB, all the modules (remember that they are all statically loaded) and the padding, if needed.
- `tls_align`: The alignement restriction for the TLS region, which is the maximum of all the `align` fields of the TLS modules and the alignement of the TCB itself.
- `tls_cnt`: Number of TLS modules.

```
struct __libc {
  char can_do_threads;
  char threaded;
  char secure;
  volatile signed char need_locks;
  int threads_minus_1;
  size_t *auxv;
  struct tls_module *tls_head;
  size_t tls_size, tls_align, tls_cnt;
  size_t page_size;
  struct __locale_struct global_locale;
};
```

Code snippet 4: The `libc` object

```
struct tls_module {
  struct tls_module *next;
  void *image;
  size_t len, size, align, offset;
};
```

Code snippet 5: The `tls_module` object

Similarly, there is a `__sysinfo` object where the loader should store the value of the `AT_SYSINFO` from the auxiliary vector, but this is ignored in 64 bits mode [6].

2.2.4 Observations

It is interesting to note that when Musl creates a thread, it sets up the TLS region at the top of the stack. While this may be peculiar at first, it makes sense as the TLS region and the stack are the only memory region specific to a thread, the text and data segment are shared with all the other threads of the process. However, this could be a security concern as it makes the reference stack canary stored in the TCB may be accessed by overflowing the stack.

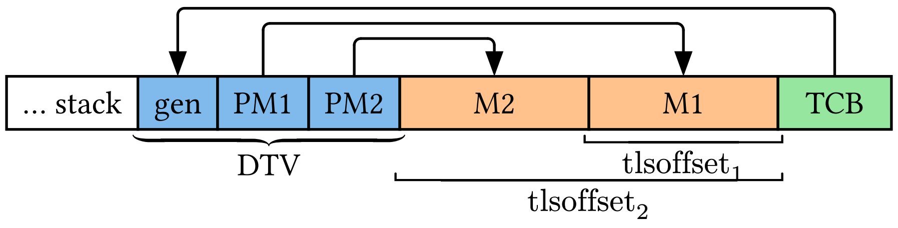

Figure 2: TLS region in Musl

3 Implementation

To fully handle TLS in the Fold architecture, I added four modules. The first one is not specific to TLS nor to SystemV (hence placed in `fold/src/`) while the three others form a chain to identify and process all TLS-related info in the ELF.

3.1 Musl

This first module has a simple purpose: locating the objects discussed in Section 2.2.3 (`__libc` and `__sysinfo`) as they are needed when setting up the TCB (see Code snippet 3). Since these objects need to be written to, the module cannot simply clone them and put them in the shared memory, neither can it store mutable references. Hence, the module stores a small struct indicating in which segment and at which offset the object is stored (see Code snippet 6).

```
pub struct MuslObjectIdx<T> {
    range: Range<usize>,
    segment: Handle<Segment>,
    data: PhantomData<T>,
}
```

Code snippet 6: Structure used to store the location of a Musl object

This structure then exposes two functions `get` and `get_mut` that return (mutable) references to the underlying object. This way, any module that wants to access a Musl Object can do so by first retrieving and cloning the `MuslObjectIdx` from the shared map, and then using it to borrow part of the manifold’s struct. Under the hood, these functions uses the `zerocopy` crate[7] which allows to safely convert a `&[u8]` into a `&T` or a `&mut [u8]` into `&mut T`, and vice-versa.

3.2 Collection

Now let's look into the TLS-related modules. The role of `TlsCollector` is to identify all the TLS modules and generate their metadata: Module ID and tlsoffset. For the former, it checks whether the object containing the module is the initial ELF (using the new `INITIAL_ELF_KEY` shared map entry) or uses a counter to generate new IDs otherwise. The formula to compute the offset of a module is shown in Figure 3 (TLS specification §3.4.6[1]). The module then stores a `Vec` containing all the TLS modules (Code snippet 7) into the manifold's shared map, as well as the individual elements in their respective object's shared map.

```
pub struct TlsModule {
    pub id: usize,
    pub tls_offset: usize,
    pub object: Handle<Object>,
    pub segment: Handle<Segment>,
}
```

Code snippet 7: Output of the `TlsCollector` module

$$\begin{cases} \text{tlsoffset}_1 = \text{round}(\text{tls\_size}_1, \text{tls\_align}_1) \\ \text{tlsoffset}_{n+1} = \text{round}(\text{tlsoffset}_n + \text{tls\_size}_{n+1}, \text{tls\_align}_{n+1}) \end{cases}$$

$$\text{with round}(x, y) \overset{\text{def}}{=} y \cdot \left\lceil \frac{x}{y} \right\rceil$$

Figure 3: Computation of `tlsoffset`

3.3 Allocation

TODO: maybe split into TlsAllocation and TlsMusl

This module is the heart of the implementation of TLS. Its role is to allocate and initialize the memory region that will store the TCB, DTV and all the modules (following Musl's implementation, all modules are statically allocated). It begins by allocating a new memory region using `mmap`, then fills it according to Figure 4. It also updates the `libc` object with all the data related to TLS; the `tls_head` list is constructed by this module and stored in the shared map to be exposed to the runtime (see also Section 6).

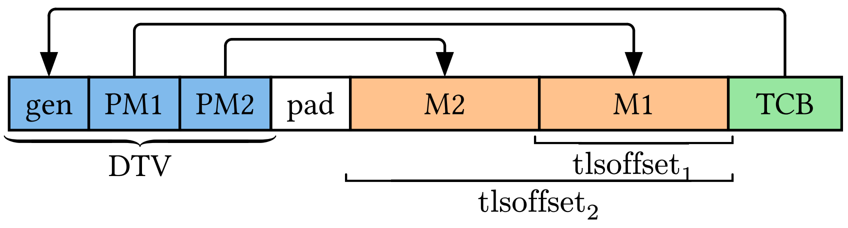

Figure 4: Main thread's TLS region in Fold

Particular care must be taken when computing the addresses, as each components have their own alignment restriction: the TCB and DTV are aligned on 8 bytes, and each module may request a specific alignment. To cover this, an optional padding region is added between the DTV and the modules (see Figure 4), such that the DTV is at the start of a page, and the TCB can be aligned on the maximum alignment requirement of the TCB and all the modules. After that, the `tlsoffset` formula (Figure 3) ensures that all the modules are properly aligned.

3.4 Relocation

This last modules processes all the TLS-related relocations found in the objects. It is separated from `SysvReloc`, but it may be interesting to merge both for performance reasons; using separate modules means that the relocations will be iterated over twice.

The relocations handled by `TlsRelocator` are:

- `R_X86_64_TPOFF{32,64}`: Offset of the symbol relative to the thread-pointer, i.e. `tlsoffset + sym.value`.
- `R_X86_64_DTPMOD64`: Module ID of the TLS module containing the symbol.
- `R_X86_64_DTPOFF{32,64}`: Offset of the symbol within its TLS module, i.e. `sym.value`.
- `R_X86_64_GOTTPOFF`, `R_X86_64_TLSGD` and `R_X86_64_TLSLD` are left unimplemented for now as they require proper handling of the GOT, which is not yet implemented in Fold (see Section 4.2)

4 Discussions

4.1 All static

As discussed in Section 2.2.1, Musl allocates all TLS modules statically upon program loading/thread start. TLS specification allowed dynamic allocation such that if a thread does not access a TLS module, then it would never get loaded for that specific thread. This would avoid allocating the memory, copying the `.tdata` section and zeroing out the `.tbss` section, hence reducing both memory usage and thread startup time cost.

However, when taking a closer look at the memory size of TLS modules (Appendix B), it is clear that the optimization gained by dynamic allocation would likely be negligeable or could even backfire. Due to the small size of the modules, all those possibly used by an executable would likely fit into a single page with the TCB, hence rendering null the memory optimization side. For the time performance, we must first observe that allowing dynamic allocation shifts the cost from $O(\text{tls\_size})$ to $O(\#\text{tls\_accesses})$ since it requires a check during each calls to `__tls_get_addr`. GNU's libc improves this by marking the check as likely to yield "allocated" for branch prediction, but one may still wonder whether there is an actual gain with dynamic allocation. This is however no the focus of this work.

4.2 Global Offset Table relocations

As said in Section 3.4, the relocations using the Global Offset Table (GOT) are not implemented as Fold itself does not implement proper handling of the GOT. However, when I checked the shared object present in my system, none used this relocation. This means all these librairies (and thus probably GCC itself) instead creates the structure given to `__tls_get_addr` in place instead of inside the GOT, which is technically not compliant with the specification. I was able to observe the same on several alpine-based docker image, i.e. with librairies built with Musl's compiler.

The command in Code snippet 8 prints the number of `R_X86_64_GOTTPOFF` occurring in all the shared object of the system. Running it on the docker images `alpine`, `alpine/ansible`, `node:alpine` and `postgresql:alpine` yields 0 occurrences.

```
find / -type f -name '*.so' -exec readelf -rW {} \; \
    | grep R_X86_64_GOTTPOFF \
    | wc -l
```

Code snippet 8: Count `R_X86_64_GOTTPOFF` in all the shared objects

I was not able to find a source for why this relocation was removed, but we can observe that in the case of `libcount-pic.so` (see Section 5), the structure targeted by the `R_X86_64_DTP*` relocation is located in the GOT, so it may be that as modern compilers generate themselves the GOT for the shared object, the relocations to allocate/get the address of the GOT are not necessary anymore.

5 State of the project

On top of all the existing samples (that still execute correctly), some new samples where added to the project to test the different ways the loader has to interact with the TLS:

- `hello-threaded` is a standalone executable with two static thread-local variables, one initialized (`value`) and one left uninitialized (`id`). The executable prints these values, then assigns its id to `id` and print it again. The expected output is to have `value` always output as its value set in the code, while `id` should first be `0` then hold the thread's id. On top of that, a `count` thread-local variable defined by another shared object (`libcount.so`) incremented and printed. This is ran on 5 different threads.
- `hello-threaded-pic` is the same as `hello-threaded`, expect that `count` is now accessed through an `incr()` function defined by the same object (now `libcount-pic.so`).
- `hello-threaded-ext` is similar to `hello-threaded-pic` except that `incr()` is defined in yet another object (`libcount-ext.so`).

All these examples run without any issues, accessing the different values through various means.

6 Future work

A lot of work is still needed for Fold and its default module chain to be usable in a real environment. From the initial project[8], the processing of jump slot relocation is still the outstanding one, along with the other missing relocations and optimizations proposed.

On top of that, while working on implementing TLS, I noticed that a pattern that may appear in the development of new modules is the need to share some data (let it be actual code, information or any other memory content) with the runtime. Currently this appears when allocating the stack of the program in `SysvStart` and the `TlsModule` linked list in `TlsAlloc` (Section 3.3), which are respectively stored in a `Vec<u8>` and the shared map of the manifold. Especially for the `TlsModule` linked list, this means that if the entry was to be moved for any reason, it would likely cause a crash once the runtime needs to access that struct. It could be useful to have an object designed to persist some memory once the control is handed over to the runtime. It should also be coupled with an allocator to allow using the existing primitives in Rust such as `Box` or `Vec`.

7 References

[1] U. Drepper, “ELF Handling For Thread-Local Storage.” [Online]. Available: https://www.akkadia.org/drepper/tls.pdf

[2] “Musl libc.” [Online]. Available: https://musl.libc.org/

[3] Chao-tic, [Online]. Available: https://chao-tic.github.io/blog/2018/12/25/tls

[4] MaskRay, [Online]. Available: https://maskray.me/blog/2021-02-14-all-about-thread-local-storage

[5] [Online]. Available: https://android.googlesource.com/platform/bionic/+/HEAD/docs/elf-tls.md

[6] “x86-specific ELF Auxiliary Vectors.” [Online]. Available: https://www.kernel.org/doc/html/next/x86/elf_auxvec.html

[7] [Online]. Available: https://docs.rs/zerocopy/0.8.31/zerocopy/

[8] L. Mermod and N. Terrier, [Online]. Available: https://github.com/epfl-dcsl/fold

A Musl's TCB

```c
struct pthread {
  /* Part 1 -- these fields may be external or
   * internal (accessed via asm) ABI. Do not change. */
  struct pthread *self;
#ifndef TLS_ABOVE_TP
  uintptr_t *dtv;
#endif
  struct pthread *prev, *next; /* non-ABI */
  uintptr_t sysinfo;
#ifndef TLS_ABOVE_TP
#ifdef CANARY_PAD
  uintptr_t canary_pad;
#endif
  uintptr_t canary;
#endif

  /* Part 2 -- implementation details, non-ABI. */
  int tid;
  int errno_val;
  volatile int detach_state;
  volatile int cancel;
  volatile unsigned char canceldisable, cancelasync;
  unsigned char tsd_used:1;
  unsigned char dlerror_flag:1;
  unsigned char *map_base;
  size_t map_size;
  void *stack;
  size_t stack_size;
  size_t guard_size;
  void *result;
  struct __ptcb *cancelbuf;
  void **tsd;
  struct {
    volatile void *volatile head;
    long off;
    volatile void *volatile pending;
  } robust_list;
  int h_errno_val;
  volatile int timer_id;
  locale_t locale;
  volatile int killlock[1];
  char *dlerror_buf;
  void *stdio_locks;

  /* Part 3 -- the positions of these fields relative to
   * the end of the structure is external and internal ABI. */
#ifdef TLS_ABOVE_TP
  uintptr_t canary;
  uintptr_t *dtv;
#endif
};
```

Code snippet 9: Musl's Thread Control Block

B TLS module sizes

Table 1 shows the size of the TLS modules of all the shared objects present in the `alpine/ansible:2.20.0` docker image. It was obtained by running the script in Code snippet 10 in the image. The average for the size in the file (**F**) and the size in memory (**M**) is respectively 0.72 and 2.16 bytes, both means are at 0.

Object	F	M	Object	F	M
_asyncio.cpython-312-x86_64.so	0	0	_bcrypt.cpython-312-x86_64.so	32	72
_bisect.cpython-312-x86_64.so	0	0	_blake2.cpython-312-x86_64.so	0	0
_bz2.cpython-312-x86_64.so	0	0	_cffi_backend.cpython-312-x86_64.so	0	4
_codecs_cn.cpython-312-x86_64.so	0	0	_codecs_hk.cpython-312-x86_64.so	0	0
_codecs_iso2022.cpython-312-x86_64.so	0	0	_codecs_jp.cpython-312-x86_64.so	0	0
_codecs_kr.cpython-312-x86_64.so	0	0	_codecs_tw.cpython-312-x86_64.so	0	0
_contextvars.cpython-312-x86_64.so	0	0	_crypt.cpython-312-x86_64.so	0	0
_csv.cpython-312-x86_64.so	0	0	_ctypes.cpython-312-x86_64.so	0	0
_ctypes_test.cpython-312-x86_64.so	0	0	_curses.cpython-312-x86_64.so	0	0
_curses_panel.cpython-312-x86_64.so	0	0	_datetime.cpython-312-x86_64.so	0	0
_dbm.cpython-312-x86_64.so	0	0	_decimal.cpython-312-x86_64.so	0	0
_elementtree.cpython-312-x86_64.so	0	0	_hashlib.cpython-312-x86_64.so	0	0
_heapq.cpython-312-x86_64.so	0	0	_json.cpython-312-x86_64.so	0	0
_lsprof.cpython-312-x86_64.so	0	0	_lzma.cpython-312-x86_64.so	0	0
_md5.cpython-312-x86_64.so	0	0	_multibytecodec.cpython-312-x86_64.so	0	0
_multiprocessing.cpython-312-x86_64.so	0	0	_opcode.cpython-312-x86_64.so	0	0
_pickle.cpython-312-x86_64.so	0	0	_posixshmem.cpython-312-x86_64.so	0	0
_posixsubprocess.cpython-312-x86_64.so	0	0	_queue.cpython-312-x86_64.so	0	0
_random.cpython-312-x86_64.so	0	0	_rust.abi3.so	32	96
_sha1.cpython-312-x86_64.so	0	0	_sha2.cpython-312-x86_64.so	0	0
_sha3.cpython-312-x86_64.so	0	0	_socket.cpython-312-x86_64.so	0	0
_sodium.abi3.so	0	0	_speedups.cpython-312-x86_64.so	0	0
_sqlite3.cpython-312-x86_64.so	0	0	_ssl.cpython-312-x86_64.so	0	0
_statistics.cpython-312-x86_64.so	0	0	_struct.cpython-312-x86_64.so	0	0
_testbuffer.cpython-312-x86_64.so	0	0	_testcapi.cpython-312-x86_64.so	0	0
_testclinic.cpython-312-x86_64.so	0	0	_testimportmultiple.cpython-312-x86_64.so	0	0
_testinternalcapi.cpython-312-x86_64.so	0	0	_testmultiphase.cpython-312-x86_64.so	0	0
_testsinglephase.cpython-312-x86_64.so	0	0	_xxinterpchannels.cpython-312-x86_64.so	0	0
_xxsubinterpreters.cpython-312-x86_64.so	0	0	_xxtestfuzz.cpython-312-x86_64.so	0	0
_yaml.cpython-312-x86_64.so	0	0	_zoneinfo.cpython-312-x86_64.so	0	0
afalg.so	0	0	array.cpython-312-x86_64.so	0	0
audioop.cpython-312-x86_64.so	0	0	binascii.cpython-312-x86_64.so	0	0
capi.so	0	0	cmath.cpython-312-x86_64.so	0	0
fcntl.cpython-312-x86_64.so	0	0	grp.cpython-312-x86_64.so	0	0
legacy.so	0	0	libbfd-2.45.1.so	0	20

libdep.so	0	0	libopcodes-2.45.1.so	0	0
libpython3.so	0	0	loader_attic.so	0	0
math.cpython-312-x86_64.so	0	0	mmap.cpython-312-x86_64.so	0	0
ossaudiodev.cpython-312-x86_64.so	0	0	padlock.so	0	0
pyexpat.cpython-312-x86_64.so	0	0	readline.cpython-312-x86_64.so	0	0
resource.cpython-312-x86_64.so	0	0	select.cpython-312-x86_64.so	0	0
spwd.cpython-312-x86_64.so	0	0	syslog.cpython-312-x86_64.so	0	0
termios.cpython-312-x86_64.so	0	0	unicodedata.cpython-312-x86_64.so	0	0
xxlimited.cpython-312-x86_64.so	0	0	xxlimited_35.cpython-312-x86_64.so	0	0
xxsubtype.cpython-312-x86_64.so	0	0	zlib.cpython-312-x86_64.so	0	0

Table 1: TLS module sizes in `alpine/ansible:2.20.0`

,

```bash
#!/bin/bash
if ! command -v readelf &> /dev/null; then
  echo "Missing readelf in PATH"
  exit 1
fi

get-segment-field() {
  readelf -lW $1 2> /dev/null | grep TLS | xargs echo | cut -d ' ' -f $2
}

for file in $(find / -type f -name '*.so'); do
  fs=$(($(get-segment-field $file 5)))
  ms=$(($(get-segment-field $file 6)))

  fs=${fs:-0}
  ms=${ms:-0}

  echo $file,$fs,$ms
done
```

Code snippet 10: Measure TLS module sizes of all shared object in the system