Matrix Deterministic Memory

Technical Reference

Table of Contents

Overview

Memory management issues in C code libraries can often be a concern for those considering the use of third-party software. Problems resulting from memory leaks and buffer overruns are infamously difficult to reproduce and debug. These issues become especially critical in the resource-constrained markets for which Matrix security products are designed. INSIDE Secure has alleviated these memory management concerns with the implementation of deterministic memory allocation. Throughout this document, the term **psMalloc** will be used in reference to this feature.

The psMalloc implementation enables the user to perform a one-time allocation of memory from the system that future allocations will draw from. This mechanism prevents the possibility of memory leaks and greatly diminishes the chance of a fault resulting from an overrun. In addition, the implementation gives the user complete control over the amount of memory being allocated and also improves speed by eliminating expensive calls to native memory routines such as malloc.

psMalloc implements a very fast and optimized version of the standard malloc library, with the addition of pools, defragmentation, alignment, and checks for out-of-memory conditions.

Static memory buffers may also be used in conjunction with psMalloc support. The static memory feature does not require a system malloc call at all. One-time static memory buffers in the application space are carved off at initialization and all future allocations are taken from this static buffer.

The psMalloc functionality is standalone but many examples and discussions in this document are in reference to the MatrixSSL product in which built-in psMalloc support has been included.

Memory Pools

The basis for the psMalloc implementation is a **memory pool.** A memory pool is an allotment of system memory that subsequent smaller allocations will draw from. There may be any number of user created memory pools. Information on how to create a memory pool can be found in the Application Programming Interface section of this document.

Why Use a Memory Pool?

Memory pools enhance malloc functionality by associating memory usage with a specific life cycle or functional area. For example, when designing a server a natural life cycle for memory usage is the lifetime of a single client connection. A pool can be created when the connection is accepted, used to allocate memory associated with that connection, and be deleted when the session is closed. Closing a pool frees all memory blocks within a pool, even if they have not been explicitly freed. Memory leaks often occur due to error conditions that are difficult to reproduce, so closing a pool guarantees that even ‘leaked’ memory is freed. The Apache Web Server uses a similar per-connection pool allocation method.

Memory pools also help to identify where memory is used, and how to minimize its usage. psMalloc returns NULL from functions that allocate memory if not enough memory is available. It is important to always check the return values of these functions to gracefully handle out-of-memory errors.

Limitations of Memory Pools

Because a memory pool is specified with a fixed size, it can rarely be sized as small as the exact amount of memory that is required. The pool will typically be sized with larger free area than required. This, combined with the overhead of the pool management itself will typically require more memory than if pools were not used.

Using a single pool for the entire application or system can minimize pool overhead. This removes most of the benefit of memory leak resilience, but still provides the optimizations and fragmentation reduction of the psMalloc system.

Most memory management algorithms impose some overhead in memory per allocated block. psMalloc is no exception. Each allocation within a pool requires a header of 4 bytes + 2 * sizeof(void *). On 32 bit platforms this amounts to 12 bytes of overhead. Additionally, allocation requests smaller than 16 bytes as defined MIN_BLOCK_SIZE in psmalloc.h, are rounded up to this value. psMalloc also imposes a limit of 216 - 1 bytes as the maximum size of an allocated block within a pool, due to the way auto-defragmentation is implemented.

Memory Allocation Strategy and Defragmentation

psMalloc implements a very efficient memory allocation algorithm. Memory is allocated from pools using queues of defined sizes (cached queues) and from lists of arbitrary sizes. Cached queues are defined with the *psAddPoolCache* API. Memory blocks of other sizes are stored in sorted lists. For any two cache sizes, there is a sorted list of block sizes that range in size between the two cache sizes. In this way, the cache queues not only provide fast allocation of blocks of a particular size, they also control the length of the arbitrary sized queues.

In practice, neither the cached queues nor the arbitrary queues grow very long because memory blocks are automatically merged with adjacent blocks when they are freed. This defragmentation is very quick, and optimal for constrained memory applications.

psMalloc also uses ‘magic’ values in the headers of allocated and free memory to help detect memory overruns or doubly freed memory.

Tuning Memory Pools

It is essential that the pool size is set large enough so that available memory will never be exceeded by a call to *psMalloc* . This will require the user to run the application under the most heavy use cases to determine the correct balance of available memory and the totalUsed statistic. In the case of the MatrixSSL handshake pool, for example, this will mean testing SSL connections with the longest supported certificate chains and key sizes.

To help with tuning, the user can activate the built-in statistics to profile the memory pools. Enable the MEM_STATS define in the psMalloc.h file to produce memory pool trace. These statistics can be used to help determine the overall pool size and the memory cache sizes. Memory statistics can be shown at any time using the *psShowPoolStats* API. Stats are also shown when a pool is closed; here is a sample of the trace provided when *psClosePool* is called.

```
Pool TMP_PKI:
size 6144
remaining 6144
highWater 3304
totalUsed 0
numSplit 75
numMerge 75
                  Size  Split Count Water
C block[00]       16    40    5     5
C block[01]       64    11    2     1
C block[02]       72    7     12    12
C block[03]       136   0     8     8
C block[04]       144   4     41    39
block[05]         152   4     2     1
C block[06]       192   0     5     5

C - Cached block
0 % Memory in Use
Memory Currently In-use: 0 bytes
Pool potentially can be shrunk by: 2672 bytes
```

size indicates the number of bytes of memory in pool storage. It is the value passed in to psOpenPool, minus sizeof(psPool_t).

remaining is the current amount of memory in bytes that is unallocated in the pool.

highWater indicates the largest total amount of memory allocated within the pool at any moment in time. It does not include the per-block header overhead.

totalUsed shows the total amount of memory in use by allocations within the pool. This number includes the per-block header overhead.

numSplit and **numMerge** indicate the number of times blocks have been split into smaller blocks (fragmented) and how many times blocks have been merged (de-fragmented).

The remaining rows show the statistics for each allocation size in the pool.

Size is the size requested by a call to psMalloc.

Split is the number of times a block of this size was split from another block, as opposed to coming off a cached list size.

Count is the total number of times that size was requested from the pool.

Water number is the highest number of simultaneous allocations for that given size.

If a **C** precedes the row for a size, that size is a memory cache queue size.

The next two stats show the amount of memory currently in use. **Memory Currently In-use** should always be 0% when closing a pool to indicate there is no memory leak (although it isn't a true memory leak because the entire pool is being freed it is highly recommended each allocation have a corresponding *psFree*)

The last statistic is the amount of memory that was never used within the pool. This could indicate that the pool can be reduced in size by this amount, if the memory usage of the pool is relatively constant run-to-run.

These statistics are useful for tuning the overall size of the memory pool and for identifying allocation sizes that are candidates for reusable cache sizes. If the count number is high, that size may benefit slightly from caching.

The process for tuning memory is as follows:

1. Set the memory pool size high
2. Run the application with maximum key sizes, cert chains, etc.
3. Observe the MEM_STATS cache suggestions
4. Cache some recommended block sizes with *psAddPoolCache*
5. Re-run the application with similar inputs
6. Observe the MEM_STATS “Pool potentially can be shrunk by” field
7. Set the memory pool size a bit larger than this value

Pool sizes can also be parameterized based on what they are used for. In the above MatrixSSL example, factors affecting memory pool requirements are:

1. Size of local keys (512 – 2048 bit)
2. Size of peer RSA key from certificate (512 – 2048 bit)
3. Number and length of fields in an X.509 certificate
4. Number of chained certificates sent by peer
5. Native digit size (MP_16BIT, MP_32BIT, etc.)

Debugging Memory Access

Each pool keeps track of how many bytes are allocated at any time. If this value is not zero when the pool is closed, the software will assert, indicating that a memory leak occurred. The psMalloc module also can detect double-frees and memory overruns on free with guard bytes. In addition, trace is available showing memory statistics for each *psMalloc* and *psFree* call, and whenever a pool is closed when MEM_STATS is enabled.

Example per-malloc statistics:

```
C memAlloced=6712 pool 00420068 (+ 256)

F memAlloced=6712 pool 00420068 (- 256)

F memAlloced=6440 pool 00420068 (- 256)

M memAlloced=6440 pool 00420068 (+ 80)

F memAlloced=6264 pool 00420068 (- 256)

M memAlloced=6264 pool 00420068 (+ 80)

F memAlloced=6088 pool 00420068 (- 256)

M memAlloced=6088 pool 00420068 (+ 152)

F memAlloced=5984 pool 00420068 (- 256)

M memAlloced=5984 pool 00420068 (+ 152)

F memAlloced=5880 pool 00420068 (- 256)

H memAlloced=5024 pool 00420068 (- 840)
```

M – psMalloc call
C – psCalloc call
F – psFree call (with caching)
H – psFree call (un-cached, "Heaped")

memAlloced – total memory allocated currently (shown pre-malloc and post-free)
pool – current pool operation applies to
(+ val) – Malloc request size
(- val) – Size of freed block

For more elaborate memory allocation debugging, third party tools are available to track buffer overruns, leaks and other memory related issues. For these tools to operate correctly, USE_MATRIX_MEMORY_MANAGEMENT must be disabled. This allows the standard C library memory routines to be used and monitored by third party tools.

Static Memory Support

The standard deterministic memory feature as described above uses the system *malloc* and *free* calls to allocate the base pool, a session pool and (when executing a full handshake) a handshake pool. If these system-level APIs are not available on a given platform, or are not to be used, the alternative is to use the static memory support built into the library. The concept is identical to the standard deterministic memory feature except that instead of grabbing a one time chunk of memory with the system malloc(), a hard coded static buffer at the application layer is created for each of the three pool types.

The memory statistics will work in the same way as the standard deterministic memory and the calculations of the pool sizes may be adjusted in the same way.

Locking Memory Pools

The deterministic memory implementation does not internally handle multi-threaded locking for any pools other than the NULL pool. If multiple threads access a single pool, the user must manually implement mutex locking for access to the pool. Because the memory operations are quite efficient, a fast spin lock is the optimal solution if locking can't be avoided. In MatrixSSL, associating a memory pool per session is a natural way to avoid the overhead of locking because typically only one thread will interact with a session at any one time.

psMalloc API

By default, deterministic memory is disabled in Matrix commercial products. The compile-time *#define* that enables psMalloc is USE_MATRIX_MEMORY_MANAGEMENT which can be found in the coreConfig.h header file. If this #define is commented out, the functions for creating pools, adding caches, and freeing pools will not be available. In addition, the *psMalloc* and *psFree* routines will map to the platform specific memory routines defined at the bottom of the psmalloc.h header file (ignoring the pool parameter).

Application code will transparently include the *coreApi.h* file when the primary Matrix product header is included (ie matrixsslApi.h).

psDefineHeap

Prototype

```
void psDefineHeap(void *heap, int32 bytes);
```

Parameters

heap	input	A pointer to static or dynamic memory to be used for all subsequent allocations.
bytes	input	The size in bytes of the heap.

Return Values

None.

Description

Define a heap pointer and size, from which to allocate pools and memory. This is useful on systems with no system *malloc* implementation, or when a high performance malloc implementation is desired. If a heap is defined, system memory allocation functions including *malloc*, *calloc*, free and *realloc* are not used. This function should be called before *psOpenMalloc* (or *psCoreOpen* ,or *matrixSslOpen*) so that all memory allocation is done from the defined heap.

psOpenPool

Prototype

```
psPool_t *psOpenPool(char *name, uint32 size,
               int32 flags, void *staticAddress,
               void *userPtr);
```

Parameters

name	input	Optional string identifier for the pool being created that is output by psShowPoolStats when statistics are displayed. This is not duplicated by the pool, and should not be deleted by the caller while the pool exists.
size	input	The size in bytes of the pool
flags	input	POOL_TYPE_MALLOC or POOL_TYPE_STATIC
staticAddress	input	The memory address to the start of the static memory pool if POOL_TYPE_STATIC flags. *NULL* if POOL_TYPE_MALLOC flags
userPtr	input	Opaque pointer for integrators who are creating their own memory implementation and require a context to be saved to the psPool_t output for future calls to psMalloc, psFree, and psRealloc

Return Values

NULL	Failure
psPool_t	Success

Description

Creates a new memory pool. The returned pool pointer is used in subsequent calls to *psMalloc, psFree,* and *psRealloc*.

psAddPoolCache

Prototype

```
int32 psAddPoolCache(psPool_t *pool, uint32 size);
```

Parameters

pool	input	Pool handle from a previous call to *psOpenPool*.
size	input	The size in bytes of the desired cache size

Return Values

PS_FAILURE	Failure
PS_SUCCESS	Success

Description
Use this function to specify a memory cache queue size for a specific pool.

psMalloc

Prototype
```
void *psMalloc(psPool_t *pool, uint32 size);
```

Parameters
`pool`	input	Pool handle from a previous call to *psOpenPool*.
`size`	input	The size in bytes of the desired memory block

Return Values
`NULL`	Failure
`Memory pointer`	Success

Description
Allocate memory from a memory pool of the given size. A valid memory pointer is returned on success and NULL is returned if there is not enough memory to allocate.

psCalloc

Prototype
```
void *psCalloc(psPool_t *pool, size_t n, size_t size);
```

Parameters
`pool`	input	Pool handle from a previous call to psOpenPool.
`n`	input	The count of how many 'size' bytes to allocate. Total allocated memory will be `size * n`
`size`	input	The size in bytes of each 'n' memory chunk to allocate. Total allocated memory will be `size * n`

Return Values

`NULL`	Failure
`Memory pointer`	Success

Description
Use this routine as an alternative to *psMalloc* to allocate and initialize each byte in the pool to zero. As in the system call *calloc*, n * size should equal to the total buffer size required.

psRealloc

Prototype
```
void *psRealloc(void *ptr, size_t n, psPool_t *pool);
```

Parameters

`ptr`	input	Pointer to already-allocated memory
`n`	input	New size for the allocated memory
`pool`	input	The memory pool to which the ptr was previously allocated.

Return Values

`NULL`	Failure
`Memory pointer`	Success

Description
Use this API to reallocate a buffer to a new size, either larger or smaller. The original memory pointed to by *ptr* is copied and then zeroed internally. The pointer to the resized memory block is returned and *ptr* is invalid after this call. NULL may be returned if the pool is not large enough to hold the resized buffer (this may occur even if reducing the buffer size).

Note
This API does not exactly match the functionality of the system call *realloc* in one case. If NULL is passed as the input *ptr* to *psRealloc,* NULL is returned. In the system *realloc* call, valid memory of size 'n' is returned. *psMalloc* should be used in this case to allocate new memory.

psMallocAlign

Prototype

```
int32 psMallocAlign(psPool_t *pool,
          unsigned char **memptr,
          unsigned char **trueptr, size_t size);
```

Parameters

`pool`	input	Pool handle from a previous call to *psOpenPool*.
`memptr`	output	The 32-bit aligned memory address that the caller will use
`trueptr`	output	The true memory location to the beginning of the allocated memory. This is the address that must be freed with *psFree*
`size`	input	The size in bytes of desired memory block

Return Values

`PS_MEM_FAIL`	Failure. Internal call to *psMalloc* failed
`> 0`	Success. Number of allocated bytes (beginning at *trueptr*)

Description

A variation on the standard *psMalloc* call that returns a 32-bit aligned memory address that has also been sized to a multiple of 32-bits. This functionality can be useful in integrations with hardware crypto in which drivers require cache alignment for DMA.

This function is implemented outside the USE_MATRIX_MEMORY_MANAGEMENT define and so may be used as a replacement for a system *memalign* API.

psReallocAlign

Prototype

```
int32 psReallocAlign(unsigned char **memptr,
          unsigned char **trueptr, size_t size,
          psPool_t *pool);
```

Parameters

`memptr`	Input/output	INPUT: The current memory address that is being reallocated. OUTPUT: The new 32-bit aligned memory address that the caller will use.
`trueptr`	Input/output	INPUT: If this realloc is being performed on an aligned address that was returned from a previous call to psMallocAlign, the trueptr is the memory address that was returned from that previous call. Otherwise, set to same address as memptr. OUTPUT: The true memory location to the beginning of the allocated memory. This is address that must be freed with psFree.
`size`	input	The size in bytes of desired memory block
`pool`	input	The memory pool to which the memptr was previously allocated

Return Values

`PS_MEM_FAIL`	Failure. Internal call to psRealloc failed
`> 0`	Success. Number of allocated bytes (beginning at trueptr)

Description

A variation on the standard psRealloc call that grows or shrinks a previously returned memory address to return a 32-bit aligned memory address that has also been sized to a multiple of 32-bits. This functionality can be useful in integrations with hardware crypto in which drivers require cache alignment for DMA.

This function is implemented outside the USE_MATRIX_MEMORY_MANAGEMENT define and so may be used in any library configuration.

psFree

Prototype

```
void psFree(void *ptr, psPool_t *pool);
```

Parameters

ptr	input	Memory to be freed and returned to the pool
pool	input	The memory pool to which ptr was previously allocated

Description

Free an allocated memory block. The memory will be merged (defragmented) with free memory blocks (if available) immediately proceeding and following the free block, if not in use. The newly merged blocks will be put on the relevant free memory list based on size.

psClosePool

Prototype

```
void psClosePool(psPool_t *pool);
```

Parameters

pool	input	Pool to close. Enable MEM_STATS for statistics

Description

Close an open pool. Pass in the *pool* identifier returned from the open call. This call will call the system *free.*

MatrixSSL Pools

If USE_MATRIX_MEMORY_MANAGEMENT is enabled while compiling MatrixSSL, several memory pools throughout the life-cycle of the SSL connection are activated. This section highlights some of the memory pools used within the MatrixSSL library.

KEY_POOL

The loading of secure key information using *matrixSslNewKeys* and *matrixSslLoadRsaKeys* must occur prior to session creation so the key material is stored in a dedicated pool that is created each time *matrixSslNewKeys* is called. The pool is deleted when *matrixSslDeleteKeys* is called.

The size of the key pool is defined by SSL_KEY_POOL_SIZE in *matrixssllib.h* and has a default value of 8KB. If your implementation uses long certificate chains or numerous Certificate Authority files you may need to increase this default.

SESSION_POOL

The session pool is the second memory pool used within the MatrixSSL library and is designed to be completely transparent to the user. This pool is created at session initiation and is intelligently sized to contain storage for the primary ssl_t data type.

HANDSHAKE_POOL

The handshake pool is created internally during the SSL handshake to store temporary security and certificate information.

Like the KEY_POOL, certificate chains present the largest unknown when determining the size of the pool. If implementing a client (or a server that performs client-authentication) it may be necessary to increase the value of SSL_CERTIFICATE_MSG_OVERHEAD in *matrixssllib.h* to account for potentially large incoming certificate chains.

TMP_PKI pools

Public key encryption operations can be very memory intensive and prior to version 3.0 this memory overhead was included in the HANDSHAKE_POOL. Now, each public key operation is wrapped within a dedicated memory pool to have the shortest possible lifespan. These pools should be completely transparent to the user.

HELLO_EXT pool

The loading of CLIENT_HELLO extensions using *matrixSslNewHelloExtension* and *matrixSslLoadHelloExtension* must occur prior to session creation so the key material is stored in a dedicated pool that is created each time *matrixSslNewHelloExtension* is called. The pool is deleted when *matrixSslDeleteHelloExtension* is called.
The size of the hello extension pool is hardcoded in *matrixSslNewHelloExtension* with a value of 1KB, which can be increased if necessary.

Data buffers use MATRIX_NO_POOL by default

The one functional area of MatrixSSL in which memory pools are not used are the data buffers that are managed with the *matrixSslGetOutdata* family of APIs. Because of the large 16KB maximum record size in the SSL specification, it was not practical to create a memory pool to satisfy this requirement. Instead, the data buffer memory is internally managed to initially be a small size and grown to exactly fit the record data. Once a record is processed the data buffer will be reduced in size.
It is possible to provide a custom memory pool for these allocations by manually creating a pool with *psOpenPool* and assigning that pool to the *bufferPool* member of the *sslSessOpts_t* pointer that is passed to *matrixSslNewClientSession* or *matrixSslNewServerSession*.