musl security features

MaskRay

https://maskray.me

musl

- A libc, an implementation of the user-space side of standard C/POSIX functions with Linux extensions.
- General-purpose, not specific to the embedded domain
- Launched in 2011. Milestone 1.0 released in 2014.

Distributions using musl

- Alpine Linux, Dragora, sabotage, Talos, Void Linux
- Shipping musl as an optional package: Gentoo Linux, OpenWrt

Core Principles

- Extracted from Rich Felker's talk *Transitioning from uclibc to musl for embedded development*
- Simplicity as the core approach to size, performance, security, and maintainability
- Factoring for minimal code duplication
- Ease of navigating and understanding code
- Robustness/fail-safety
- Not depending on fancy compiler/toolchain features
- First-class status for UTF-8, non-ASCII characters

Miscellaneous

- Eager/deferred binding instead of lazy binding wiki
- v1.0.0: `PT_GNU_RELRO`: some sections are readonly after relocation resolving. `mprotect` such sections.
- v1.1.10: static pie (earlier than glibc)
- v1.1.20: `explicit_bzero`: `bzero` with a compiler barrier
- v1.1.24: `secure_getenv`: `getenv` or (set-user-ID or set-group-ID) no-op
- v1.2.2: `reallocarray`: `realloc` with multiplication overflow check (OpenBSD 5.6)

Stack Smashing Protector

- `-fstack-protector{-explicit,,-string,-all}`
- Add a secret value (canary) after local variables (before the return address) on the stack.
- On return, check whether the canary stays the same.
- `-mstack-protector-guard=` decides whether the secret is stored.

```
void foo(const char *a, const char *b) {
  char buf[16];
  strcpy(buf, a); strcat(buf, b); puts(buf);
}
```

```
foo:                                    # @foo
  # Save callee-saved registers.
  pushq %r14; pushq %rbx
  # Allocate local variables and canary.
  subq $24, %rsp
  # Copy arguments to callee-saved registers.
  movq %rsi, %r14; movq %rdi, %rsi
  # Set canary.
    movq %fs:40, %rax # -mstack-protector-guard=tls
    movq __stack_chk_guard(%rip), %rax # -mstack-protector-guard=global
  movq %rax, 16(%rsp)
  movq %rsp, %rbx
  # Main body.
  movq %rbx, %rdi; callq strcpy
  movq %rbx, %rdi; movq %r14, %rsi; callq strcat
  movq %rbx, %rdi; callq puts
  # Check canary.
    movq %fs:40, %rax # -mstack-protector-guard=tls
    movq __stack_chk_guard(%rip), %rax # -mstack-protector-guard=global
  cmpq 16(%rsp), %rax; jne .LBB0_2
  # Epilogue
  addq $24, %rsp; popq %rbx; popq %r14
  retq
.LBB0_2:
  callq __stack_chk_fail
.Lfunc_end0:
```

Stack Smashing Protector in libc

- libc has to initialize (tls) `%fs:40` or (global) `__stack_chk_guard`.
- In glibc, only one is supported (arch-specific configure-time decision). PR glibc/26817 (tls may be less secure due to lack of guard page before static TLS block)
- musl supports -mstack-protector-guard=tls and -mstack-protector-guard=global at the same time.
- v1.1.9: Allow libc itself to be built with `-fstack-protector` (earlier than glibc)

Secure mode

- set-user-ID and set-group-ID (`chmod u+s` and `chmod g+s`)
- ld.so: disallow `LD_LIBRARY_PATH`/`LD_PRELOAD`, non-absolute `$ORIGIN`, obtaining `$ORIGIN` from main executable `/proc/self/exe`
- date/time functions: disallow `TZ` file other than /etc/localtime (`TZ=:/usr/local/etc/localtime`)
- `catopen`: disallow `NLSPATH`
- Ensure fd 0/1/2 are open: a badly written set-UID program may open files (occupying fd 0/1/2) and clobber these files when writing to stdout/stderr.

mallocng

- v1.2.1: "Strong hardening against memory usage errors by the caller, including detection of overflows, double-free, and use-after-free, and does not admit corruption of allocator state via these errors."
- Mixed in-band and out-of-band metadata. Sensitive states are in the out-of-band metadata.
- Allocation granularity is 16 bytes.

Corruption of allocator state

- glibc (as of 2.31): `free(): double free detected in tcache 2` `double free or corruption (fasttop)`
- musl: crash without diagnostic

malloc implementations

- dlmalloc (Doug Lea), ptmalloc (pthread malloc), glibc, jemalloc, tcmalloc, OpenBSD omalloc, Chrome PartitionAlloc
- dlmalloc family: metadata (fd/bk/prev_size/size) in a free chunk overlay user data in an in-use chunk

Characteristics

- malloc after free: first-fit effects, similar to glibc (LIFO in tcache/fastbin; FIFO in unsorted bin). However, randomness can be trivially implemented by changing the current least significant bit heuristic. For a request of n, if the slot has $n + UNIT * (k - 1)$ bytes, `enframe` can cycle the allocation in k places, i.e. a slot can be reused without handling an identical resource identifier. This property is more useful for detecting double-free bugs than hardening.
- Main metadata is stored separately. Out-of-bounds write forges the header of the next slot, but it can hardly do harm.
- double free/invalid free: guaranteed crash due to rigorous verification. In glibc, double free of non-top chunk in fastbin cannot be detected (`double free or corruption (fasttop)`, other bins have more checks).
- No `malloc_set_zero_contents`/`malloc_set_pattern_fill_contents` (scudo). Data leak from freed memory is possible.
- No `__malloc_hook` or `__free_hook`.

```
// Singleton: ctx
struct malloc_context {
  uint64_t secret;
#ifndef PAGESIZE
  size_t pagesize;
#endif
  // When alloc_meta is invoked for the first time, initialize secret and pagesize.
  int init_done;
  unsigned mmap_counter;
  struct meta *free_meta_head;
  // [avail_meta,avail_meta_count) are available meta objs.
  struct meta *avail_meta;
  // When a new page is allocates, (4096-sizeof(meta_area))/sizeof(meta) meta objs are newly available.
  size_t avail_meta_count, avail_meta_area_count, meta_alloc_shift;
  struct meta_area *meta_area_head, *meta_area_tail;
  // [avail_meta_areas,avail_meta_areas+avail_meta_area_count) are available meta areas.
  unsigned char *avail_meta_areas;
  // Doubly linked meta list by size class,
  struct meta *active[48];
  size_t usage_by_class[48];
  uint8_t unmap_seq[32], bounces[32];
  uint8_t seq;
  // [initial brk(0)+pagesize, brk) stores available meta areas.
  uintptr_t brk;
};
```

A group contains a header (of minimum metadata) and 1 ∼ 32 slots for allocation. Each group is paired with a meta object which references back to the group.
The slot size (`stride`) is decided by the size class. The number of slots is decided by the size class and its usage.

```
struct meta_area {
  uint64_t check; // ctx.secret
  struct meta_area *next;
  int nslots; // (4096-sizeof(meta_area))/sizeof(meta)
  struct meta slots[];
};

struct meta {
  struct meta *prev, *next;
  struct group *mem;
  // Bitmask of unused slots and bitmask of freed slots. They have no intersection.
  volatile int avail_mask, freed_mask;
  // The index of the last slot, i.e. the number of slots minus 1.
  uintptr_t last_idx:5;
  uintptr_t freeable:1;
  uintptr_t sizeclass:6;
  uintptr_t maplen:8*sizeof(uintptr_t)-12;
};

struct group {
  struct meta *meta;
  unsigned char active_idx:5;
  char pad[UNIT - sizeof(struct meta *) - 1];
  // N slots. A slot starts at offset (1+stride*slot_index)*UNIT.
  unsigned char storage[];
};
```

Each chunk is user data preceded by a 4-byte header ($IB = 4$). For a requested size n, the chunk size is $n + 4$. The slot size needs to be at least as large as $n + 4$.
The canonical placement starts the user data at the slot start (offset: 0) and makes its header overlap the last 4 bytes of the previous slot. (The last 4 bytes of the current slot is reserved by the next chunk). This placement can be used for an unaligned allocation.
A chunk may have some trailing spare bytes, named `reserved`. If *reserved* $>= 5$, use the last 5 bytes to store a zero canary byte and `reserved`, verified by `get_nominal_size` (called by malloc and free). This can detect out-of-bounds writes.

```
struct chunk {
  uint8_t zero;
  uint8_t idx; // slot_index | (min(reserved,5)<<5); if nested in a larger chunk, slot_index | 6<<5
  uint16_t offset; // (p-g->mem->storage) / UNIT
  // If reserved bytes >= 5, end[-5] is 0 and *(uint32_t*)(end-4) stores reserved.
  char p[stride-IB];
};

struct chunk32 {
  uint32_t offset; // (p-g->mem->storage) / UNIT
  uint8_t non_zero;
  uint8_t idx; // slot_index | (min(reserved,5)<<5)
  uint16_t zero;
  // If reserved bytes >= 5, end[-5] is 0 and *(uint32_t*)(end-4) stores reserved.
  char p[stride-IB];
};
```

Features available in glibc but unavailable in musl

- Memory protection keys (`pkey_*`)
- AArch64 Branch Target Identification and Pointer Authentication (assembly annotation and ld.so support (`mprotext PROG_BTI`))
- AArch64 Memory Tagging Extension (including heap tagging in its malloc implementation)
- x86 IBT and SHSTK (assembly annotation and ld.so support)
- Defense in depth: `setjmp/longjmp/__cxa_atexit` address mangling

References

- Transitioning From uclibc to musl for Embedded Development, Rich Felker
- `https://dustri.org/b/security-features-of-musl.html`, Julien Voisin