Software Security 1

Administrative

Kevin Borgolte kevin.borgolte@rub.de

Tentative Lecture Schedule / Deadlines

Lectures
Wednesday 10-12

1. Oct 9 – Lecture
2. Oct 16 – Recitation
3. Oct 23 – Lecture ← First assignment due
4. Oct 30 – Recitation
5. Nov 6 – Lecture ← Second assignment due
6. Nov 13 – Recitation
7. Nov 20 – Lecture ← Third assignment due
8. Nov 27 – Recitation
9. Dec 4 – Lecture ← Fourth assignment due
10. Dec 11 – Recitation
11. Dec 18 – Lecture ← Fifth assignment due
12. Jan 8 – Lecture
13. Jan 15 – Recitation
14. Jan 22 – Lecture ← Sixth assignment due
15. Jan 29 – Recitation

Tentative

(will probably not change anymore)

Assignments

- Questions
  - Moodle or email us (softsec+teaching@rub.de)
  - If you run into issues, please report your OS, Docker version, etc.
- Assignment 4
  - 5 tasks
    - Too difficult? Too easy?
  - Due: Midnight this evening! (December 5th, 0:00 Bochum time)
- Assignment 5
  - 5 tasks
    - Focusing on heap and reversing
  - Due: December 19th, 0:00 Bochum time

Topics Today

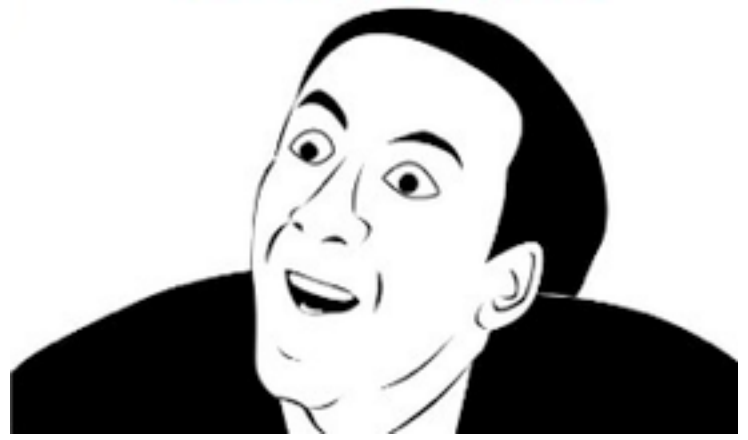
- Defensive Programming
- Heap allocator defenses
- Other heap allocators
- C++ and vtables

Software Security 1

Defensive Programming

Kevin Borgolte kevin.borgolte@rub.de

Defensive Programming?



- Writing secure software is difficult
  - Making mistakes is easy in many programming languages
  - Varied intimate background knowledge required to write robust code
    - Example: Can some code only be in some cases undefined behavior?
- Multiple approaches to encourage robust code
  - Before development begins
  - During development
  - Testing

```
int ub(void) {
  int a = 0;
  int b = 0;
  return &a < & b;
}
```

Before Development Begins

- Use a safe® programming languages, for example:
  - C is not safe
  - Modern C++ can be safe
  - Rust is safe (if unsafe is not used)
  - Python, Go, Java, etc. all have some built-in safety
- Which language is a good choice for your project depends on your performance requirements and threat model
  - Untrusted input? C is probably not a good choice
  - High performance for HPC? Input is trusted? C might be a decent choice

Before Development Begins

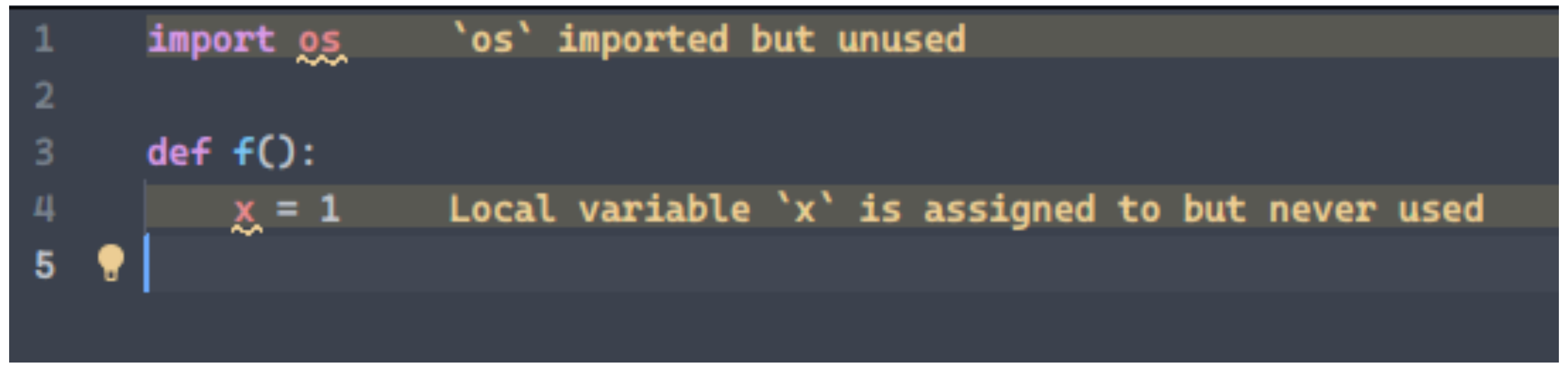
- Using a safe language does not prevent all issues
  - Their implementations might be in C/C++
    - The default Python interpreter, CPython
    - The implementations might have bugs and that cause issues
  - The libraries you use might be written in C/C++
    - NumPy
  - Unsafe code might be called through foreign function interfaces (FFI)
    - Python ctypes, or Java Native Interface (JNI)

During Development

- Be Explicit
  - Use clear variable names (duh!)
  - Check return values exhaustively
    - Some languages tell you if you don't
  - Check all assumptions
    - If they are truly *always* true, then a good compiler should optimize them away for you anyways
  - Define new types to indicate meaning (Distance vs. Balance), and help the compiler check program safety and reduce opportunities for error
- Think of your own future self looking at the same code base

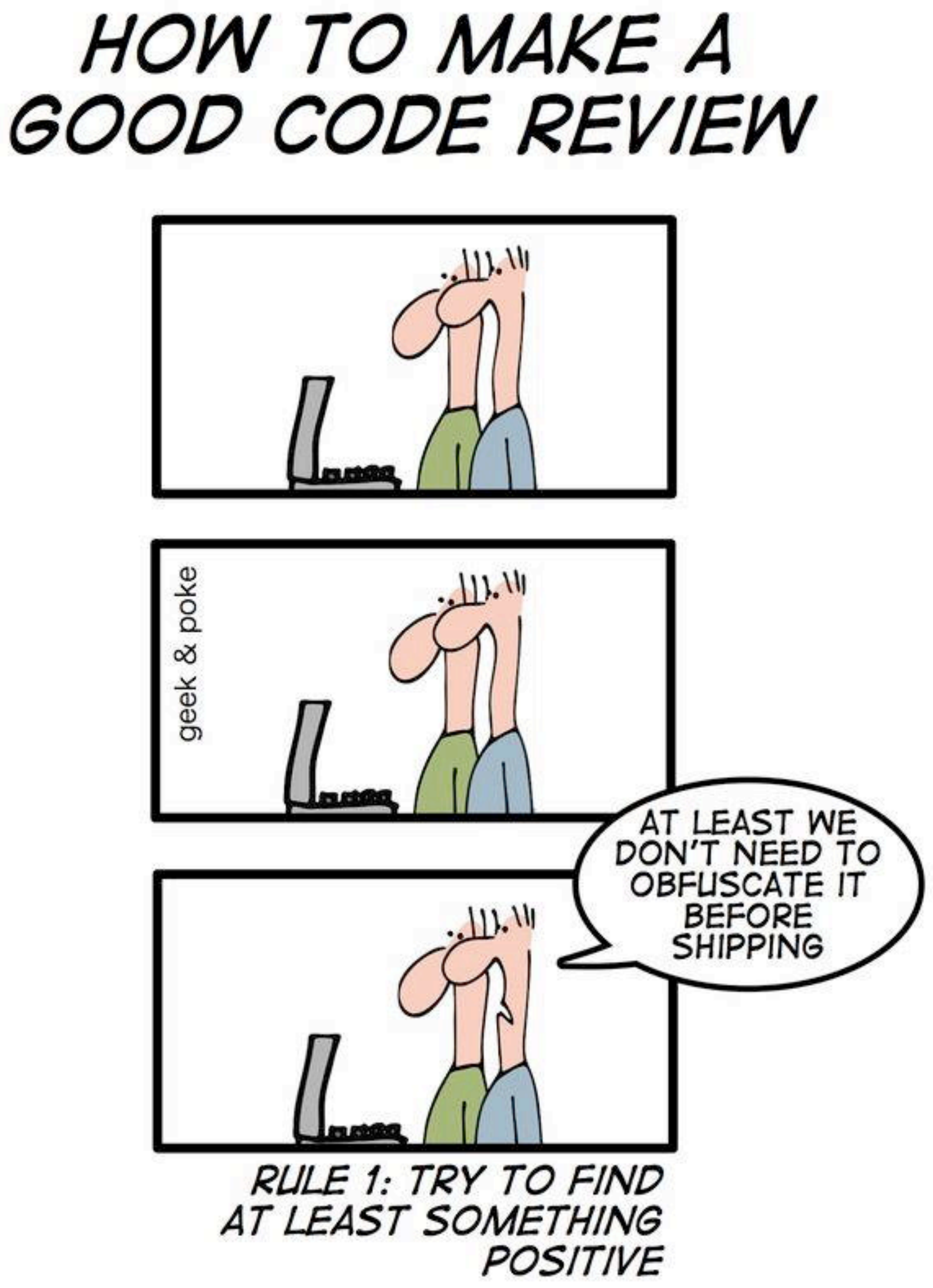
During Development

- Linting
  - Automatic code checking for inconsistencies, stylistic errors, and common bad patterns (code smells)
  - Usually used to check/enforce coding conventions for some code base
  - Can also identify some types of vulnerabilities
  - Some are integrated in development environments
  - Enforces *consistency* and *cleanliness*, and reduces *complexity*
    - Both help you spot bugs during easier



During Development

- Manual code reviews
  - Common and effective technique
  - Principle of many eyes (1-2+ others review your code)
    - Some companies require them from other teams or security experts
  - Example: Linux Kernel patch submissions via its mailing list ("Reviewed-by")
  - Discussion about formatting does not contribute to the discussion of bugs/issues (bikeshedding; focusing on trivial issues/disagreements)
    - Linting (consistency and cleanliness), helps to reduce bikeshedding



During Development

- Use safe/secure functions
  - Explicitly pass in applicable limits/sizes
- Use safe/secure libraries
  - Safe/secure versions of libraries with otherwise vulnerable functions exist
    - glibc defenses can be enabled via -D_FORTIFY_SOURCE=2 or 3
    - Introduces runtime overhead (buffer size and bounds need to checked at runtime)
    - Not necessarily 100% safe, function interfaces might be insufficient and cases can exist for which you cannot determine buffer size
      - The default is usually to "fail open" and call the insecure/unsafe function
- Verify all input and check that it is valid ("all input is evil")
- Minimize attack surface (disable functionality and drop privileges you don't need)

Resources

- More resources for safe/secure programming (focus on C)
  - Secure Programming HOWTO by D. Wheeler
    https://dwheeler.com/secure-programs/
  - SEI CERT Coding Standards (C, C++, Android, Java, Perl)
    https://wiki.sei.cmu.edu/confluence/display/seccode
  - "Secure Coding in C and C++" by R. Seacord
- They are also good intuition what can go wrong
  - Offensive/defensive are two sides of the same coin

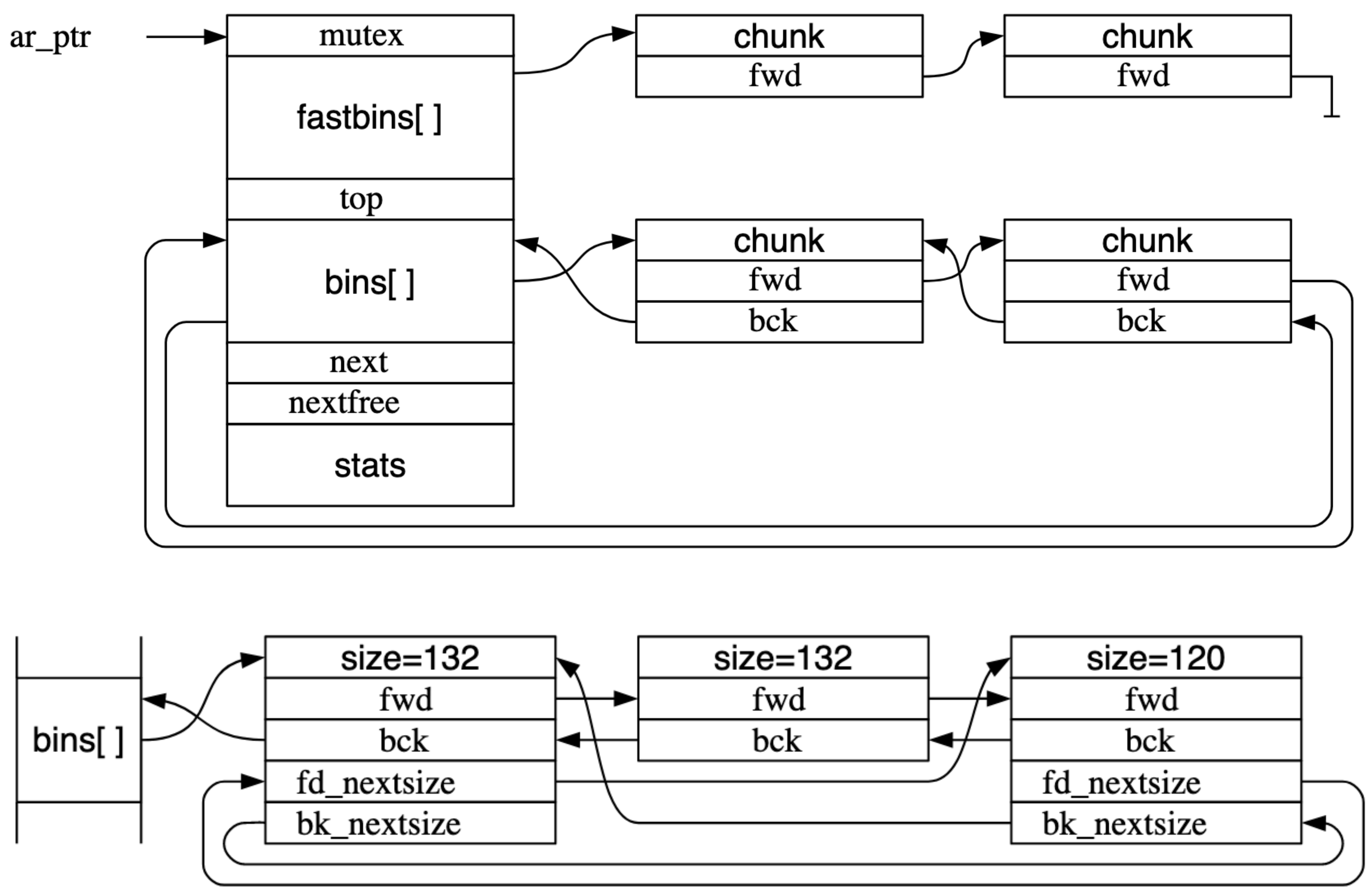
Software Security 1

Heap Defenses and Other Allocators

Kevin Borgolte
kevin.borgolte@rub.de

Pointer Mangling/Encryption

- glibc's DMA uses pointers in many places without authentication/verification
- Can we make it more difficult for attackers to abuse them?
  - Pointer Mangling/Encryption



Pointer Mangling/Encryption

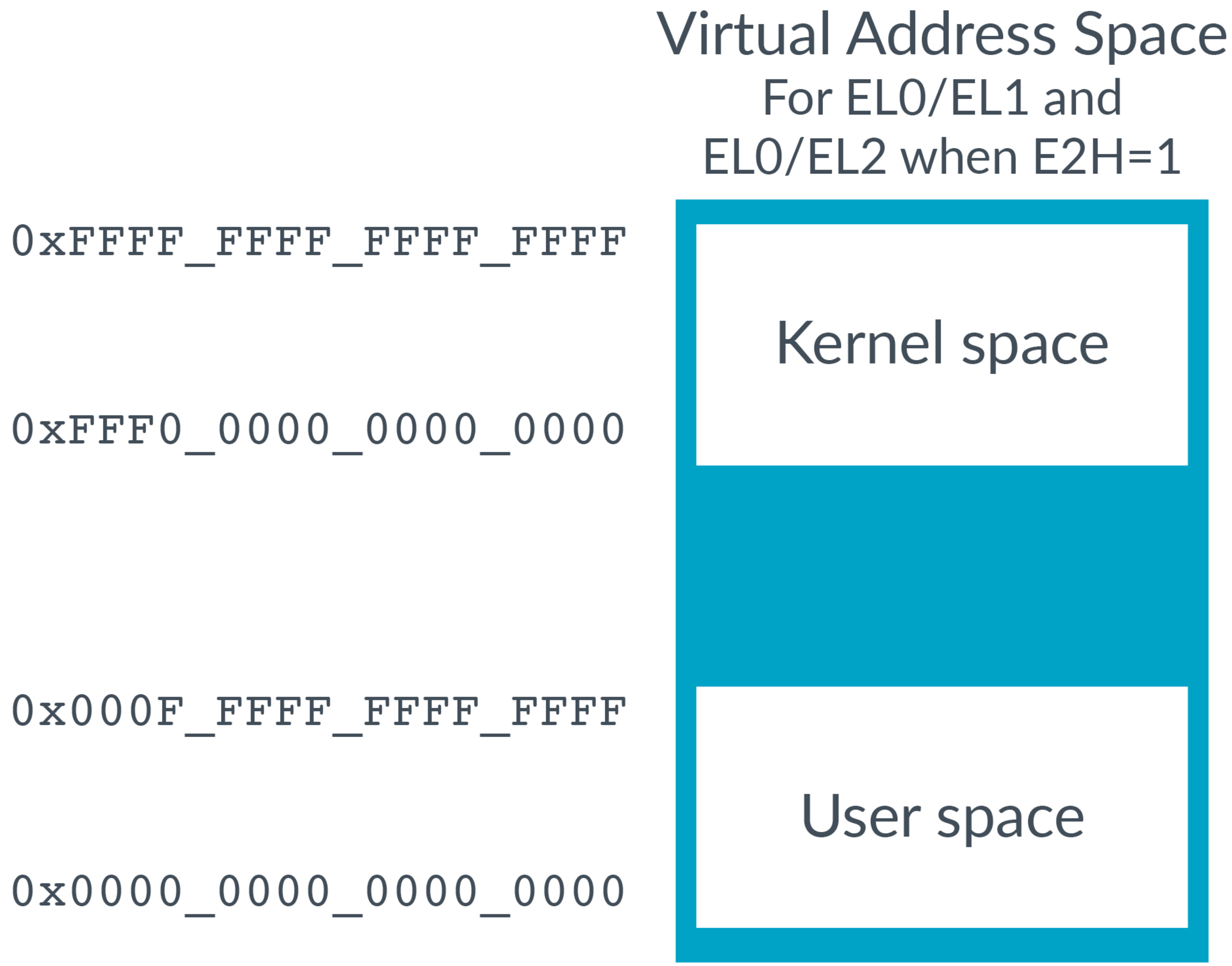
```
P  := 0x0000BA9876543210
L  := 0x0000BA9876543180

      P            =     0x0000BA9876543210
   ⊕                 ⊕
        L >> 12    =     0x0000000BA9876543
                         ------------------
P' := P⊕(L >> 12)  =     0x0000BA93DFD35753
```

- Modify all pointers, typically XOR with a random value
- Full pointer/address is "random", eliminates the issue that only part of the address is randomized (ASLR prefix)
  - Cannot just overwrite lowest bytes on stack anymore with offset difference
- More mangling/guarding than encryption
  - XOR key usually per-process value, leaking key means being able to forge pointers
  - Can be more fine grained, but value must remain somewhat stable
- Leaking pointers becomes much less useful
  - Appears to be random value, deducing stack/heap pointer from it is difficult (e.g., ASLR)
  - Cannot simply XOR two pointers to remove key
    - We only gain information about difference between the two pointers, which is rarely useful

https://sourceware.org/glibc/wiki/PointerEncryption

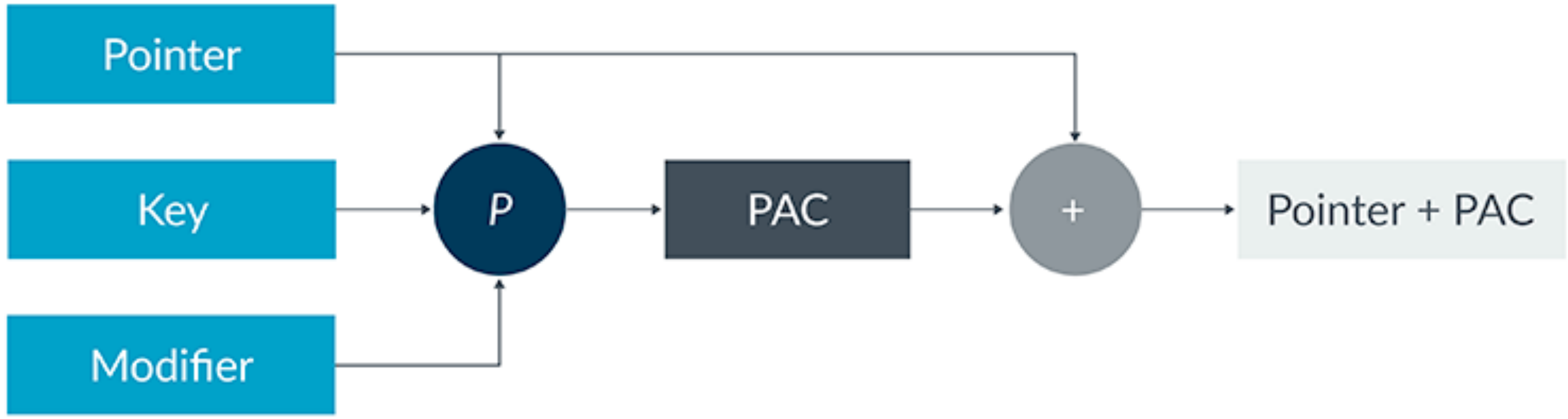
Sidebar: Pointer Authentication Code (PAC)

- Hardware feature for ARM aarch64 CPUs
- Cryptographically hash and authenticate pointers
- Use the top bits of a pointer value as the signature for the pointer because the virtual address space does not need the full 64-bit address width

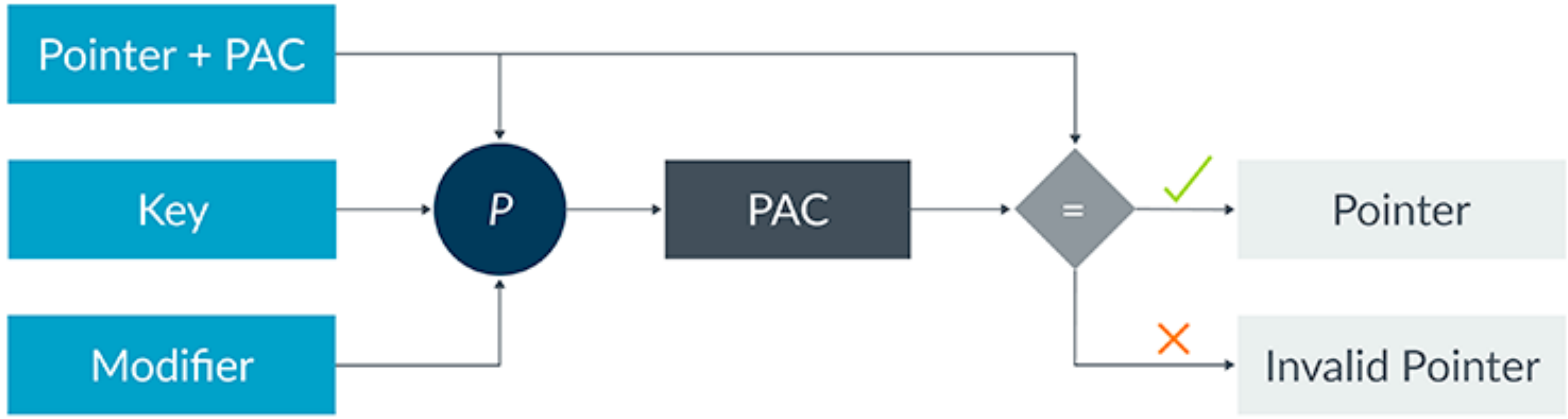


Sidebar: Pointer Authentication Code (PAC)

- Width of PAC depends on CPU model
  - 12 bits and 16 bits are common
- PAC key is only accessible at specific CPU privilege levels
  - (usually) two keys A and B exist
- Modifier is the execution context



Sign



Authenticate

Sidebar: Pointer Authentication Code (PAC): Modifier

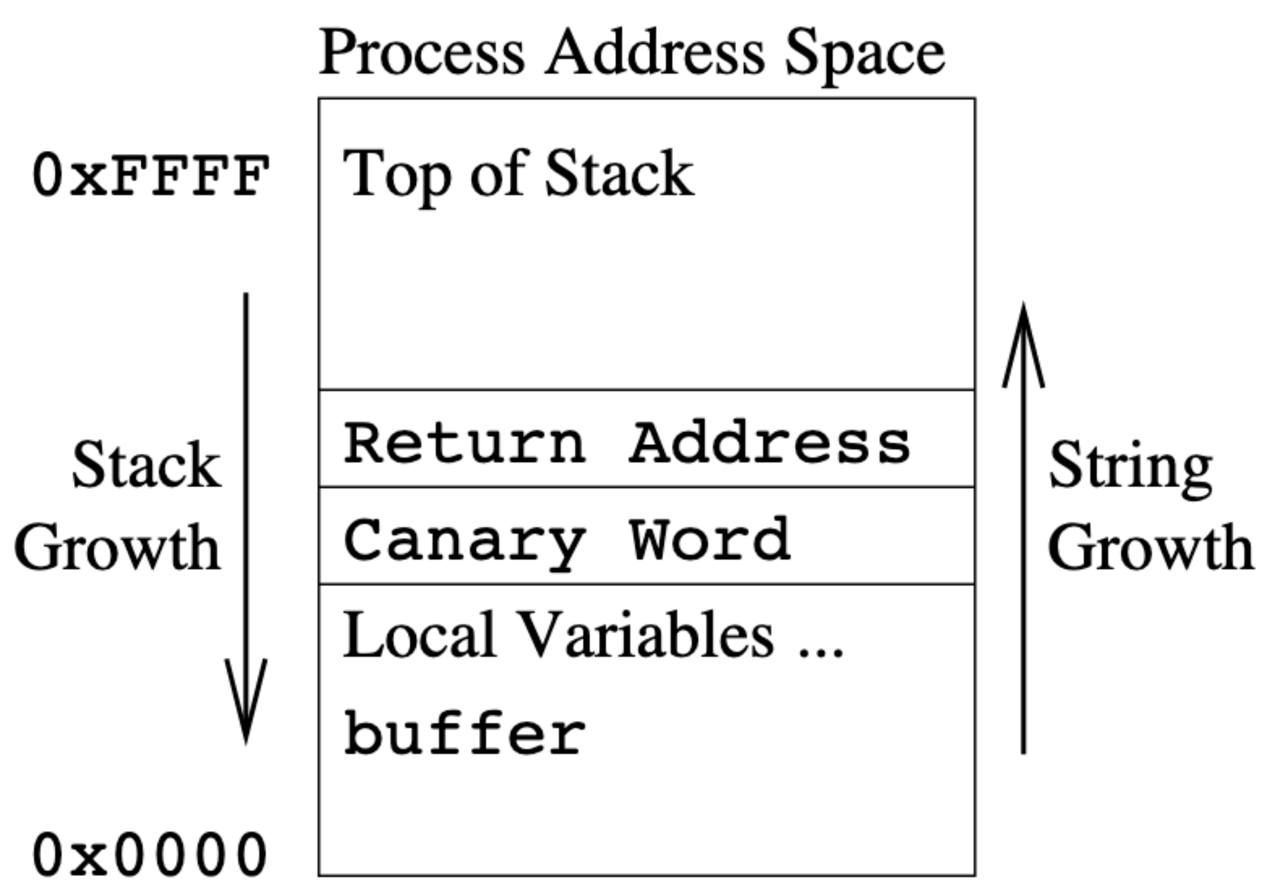
- Modifier is the execution context
  - Idea is to prevent reusing/replaying authenticated pointers
  - Stack pointer (to authenticate/verify the return address)
  - Zero to effectively not use it
  - Other registers
- Program needs to ensure that modifier does not change between pointer generation and verification

Sidebar: Pointer Authentication Code (PAC)

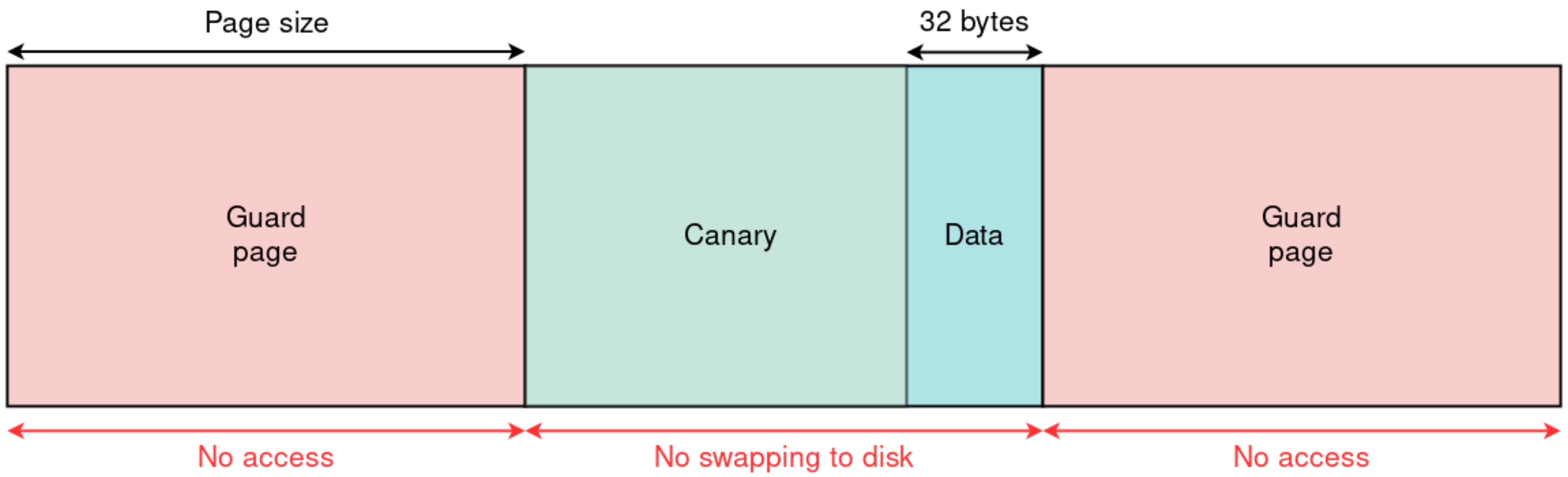
```
ldr x16, [object]
mov x17, object
movk x17, #0xd986, lsl #48
autda x16, x17
ldr x8, [x16]
```

- x16 is the authenticated address
- x17 is the modifier
  - Here a salt = 0xd986 << 48 | address
- autda (authenticate data pointer) checks the PAC signature
  - If valid, remove signature
  - If invalid, store invalid/corrupt pointer
- Pointer access works or crashes

Heap Canaries and Guard Pages



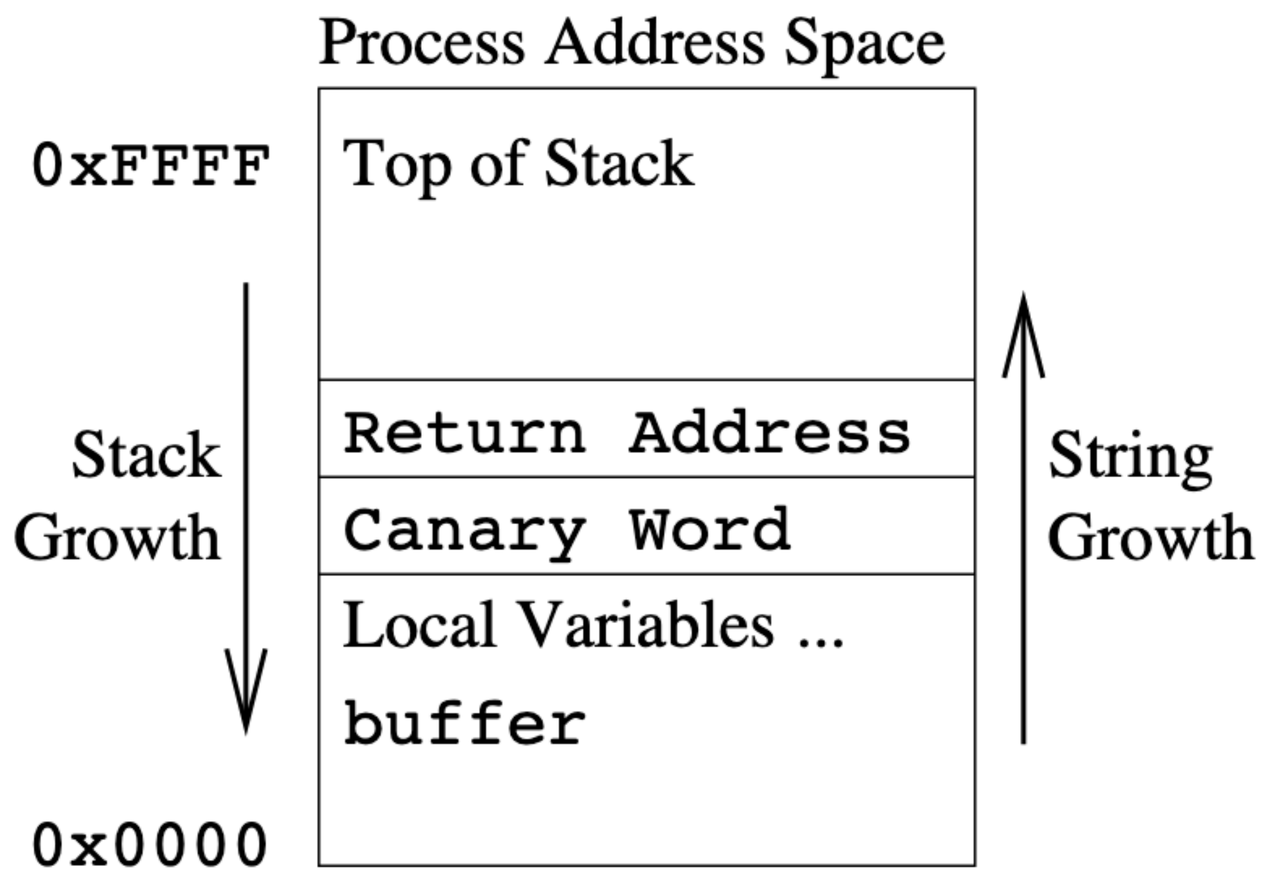
Stack Canary



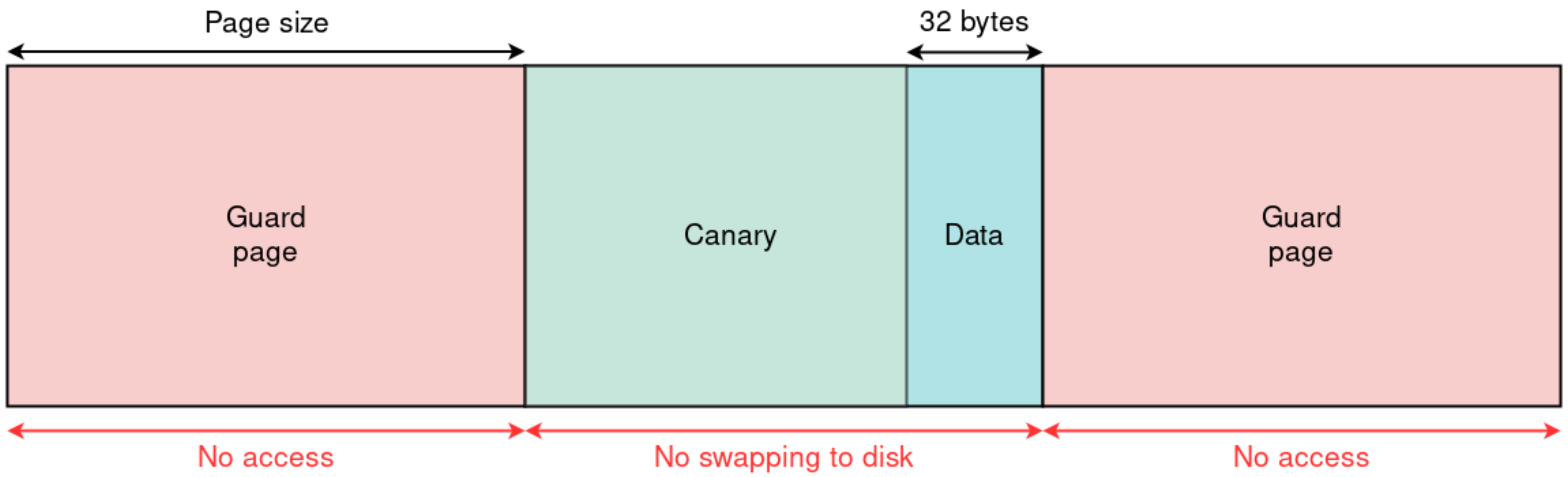
Heap Canary with Guard Pages

- Canary will be verified just like for stack
- Guard page writes trigger segfault or similar
- Both allow us to protect against heap overflows

Heap Canaries and Guard Pages



Stack Canary

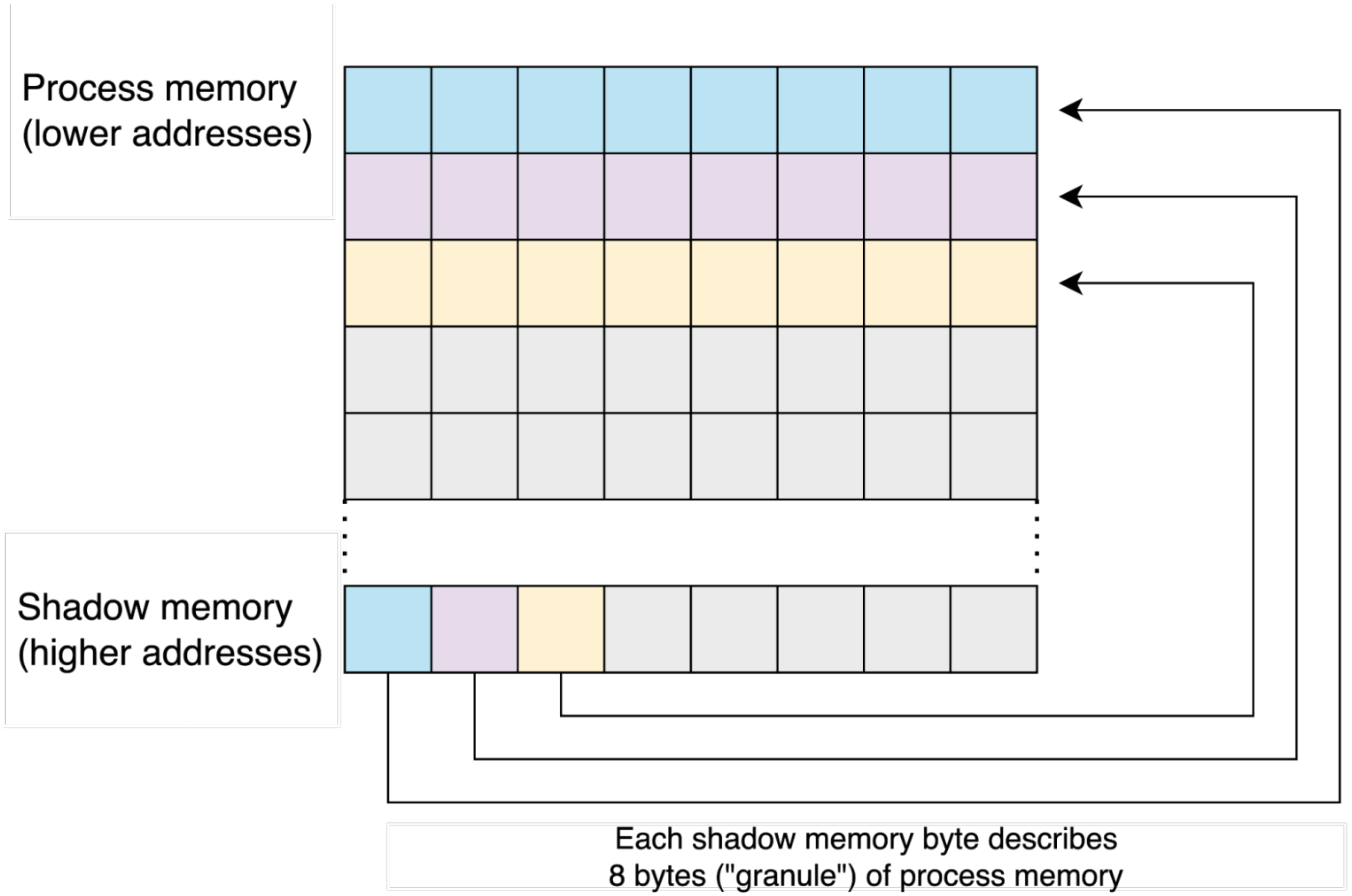


Heap Canary with Guard Pages

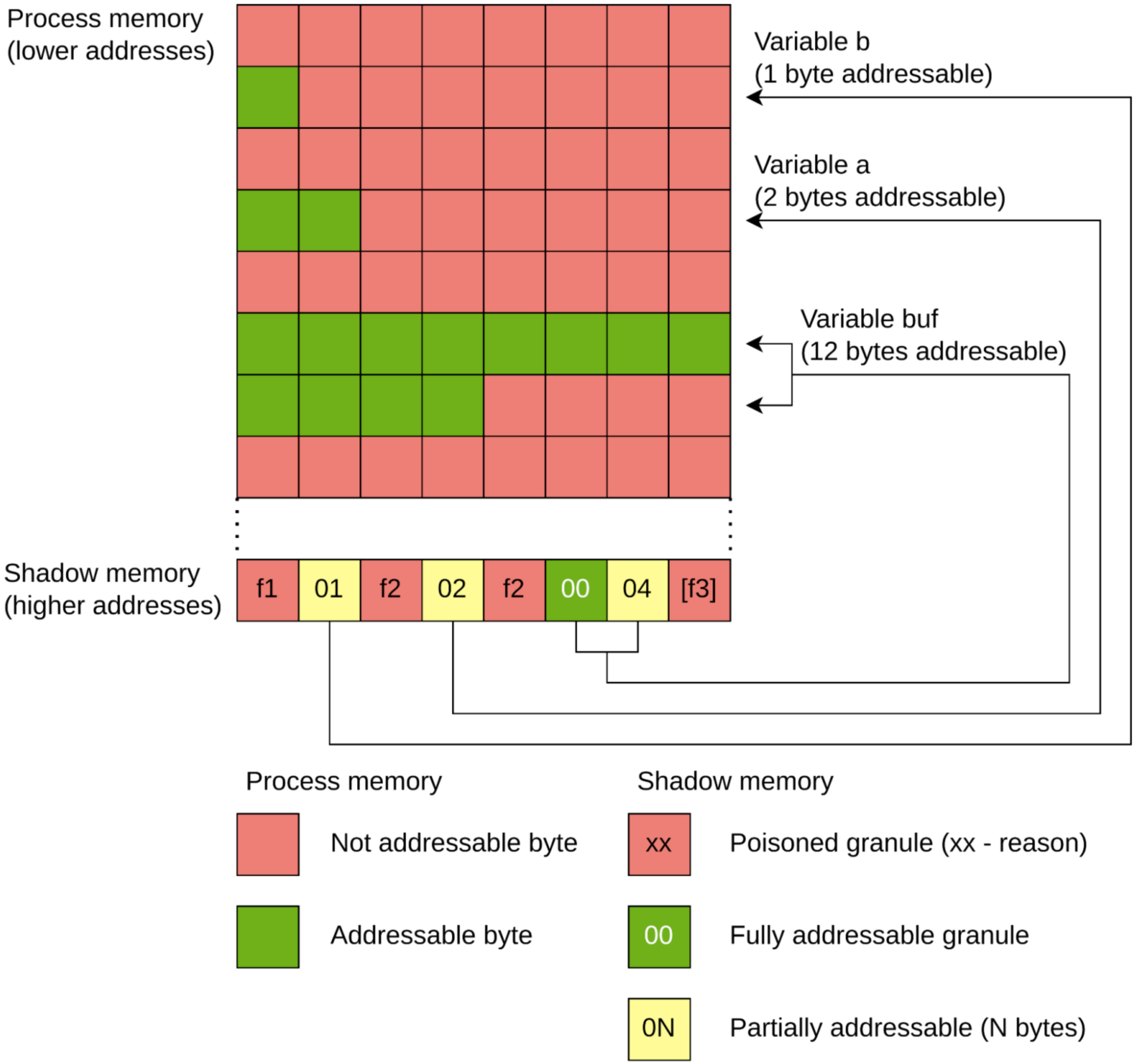
- But, they do not protect against arbitrary write-anywhere because we can skip the canary and guard pages

Address Sanitizer

- Address sanitizer (ASan) is runtime instrumentation to detect memory errors
- It uses similar concepts
  - Red zones
    - Effectively heap canaries
  - Shadow memory
    - Not unlike shadow stack
    - Typically mapped at 8 bytes granularity



Address Sanitizer



- Shadow bytes are
  - Negative: Poisoned
    - Value denotes reason, like free'd memory or specific red zones
  - Zero: fully accessible
  - Positive: Number of bytes that are accessible
    - Allows easy checks

Address Sanitizer

```
int get_element(int *a, int i) {
  return a[i];
}
```

```
int get_element(int *a, int i) {
   if (a == NULL)
      abort();

   return a[i];
}
```

```
int get_element(int *a, int i) {
   if (a == NULL)
      abort();

   region = get_allocation(a);
   if (in_stack(region)) {
      if (popped(region))
         abort();
      // ...
   }
   if (in_heap(region)) {
      // ...
   }
   return a[i];
}
```

```
int get_element(int *a, int i) {
   if (a == NULL)
      abort();

   region = get_allocation(a);
   if (in_heap(region)) {
      low, high = get_bounds(region);
      if ((a + i) < low || (a + i) > high) {
         abort();
      }
   }
   return a[i];
}
```

Finds:
Use after free (dangling pointers), heap buffer overflow, stack buffer overflow, global buffer overflow, use after return, use after scope, initialization order, memory leaks

Address Sanitizer

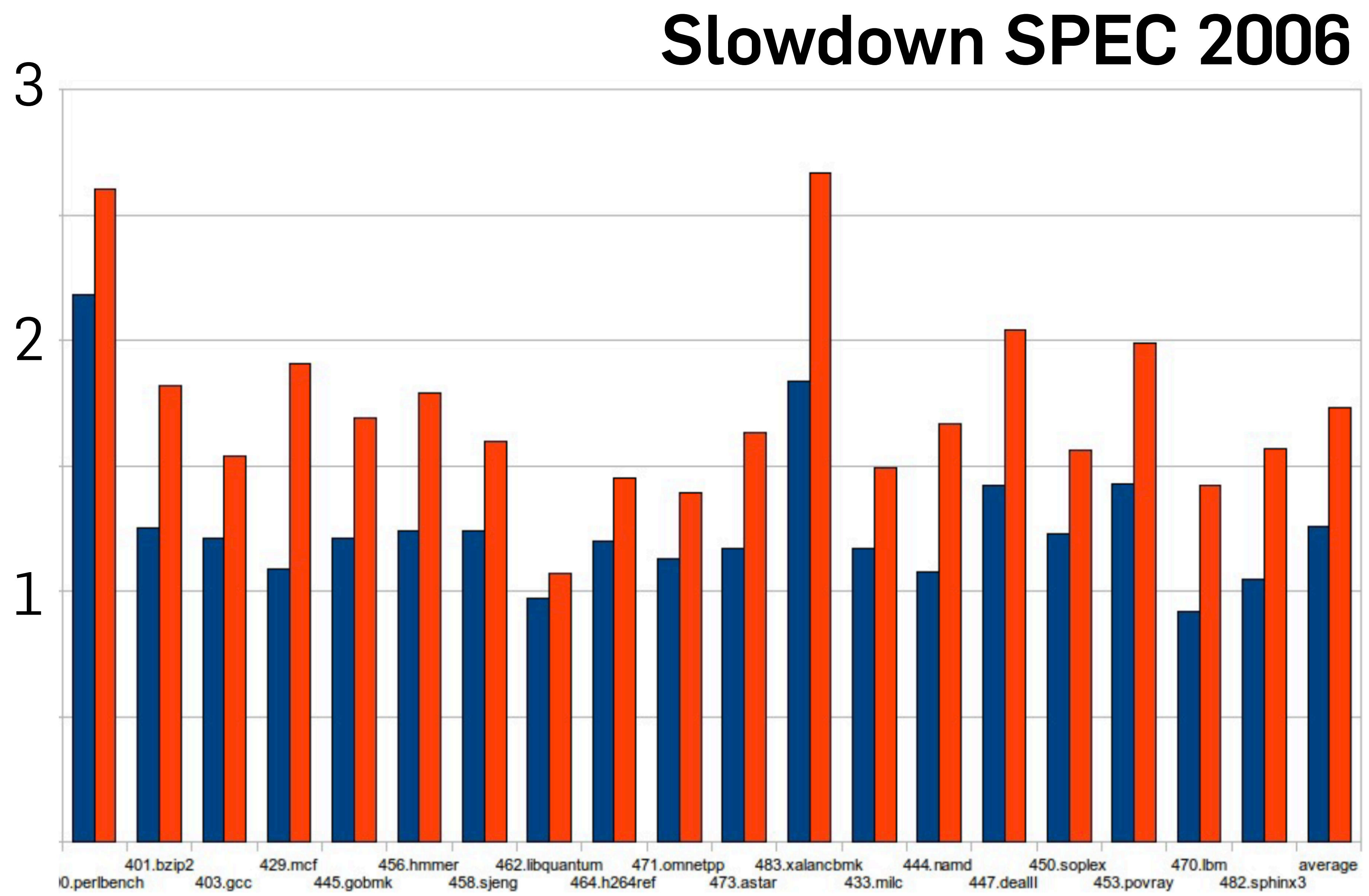
```c
int get_element(int *a, int i) {
  return a[i];
}
```

```c
int get_element(int *a, int i) {
  if (a == NULL)
    abort();

  return a[i];
}
```

```c
int get_element(int *a, int i) {
  if (a == NULL)
    abort();

  region = get_allocation(a);
  if (in_stack(region)) {
    if (popped(region))
      abort();
    // ...
  }
  if (in_heap(region)) {
    // ...
  }
  return a[i];
}
```

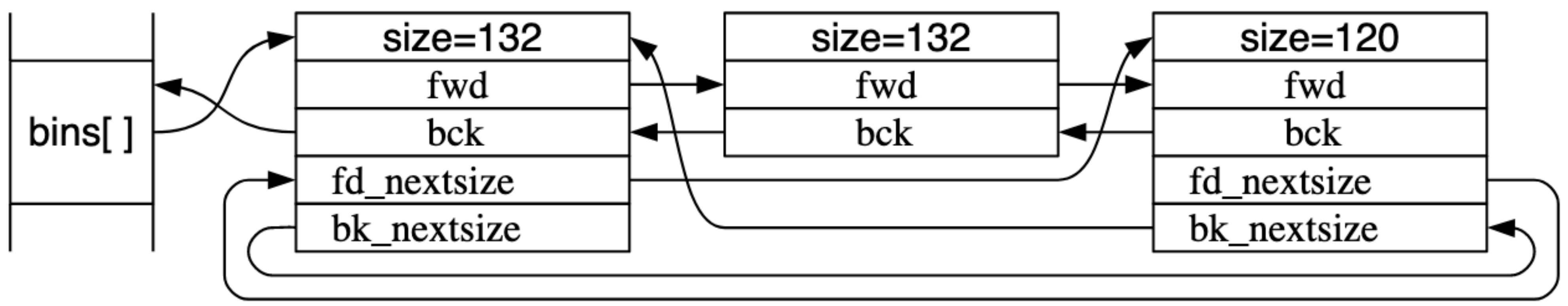
```c
int get_element(int *a, int i) {
  if (a == NULL)
    abort();

  region = get_allocation(a);
  if (in_heap(region)) {
    low, high = get_bounds(region);
    if ((a + i) < low || (a + i) > high) {
      abort();
    }
  }
  return a[i];
}
```

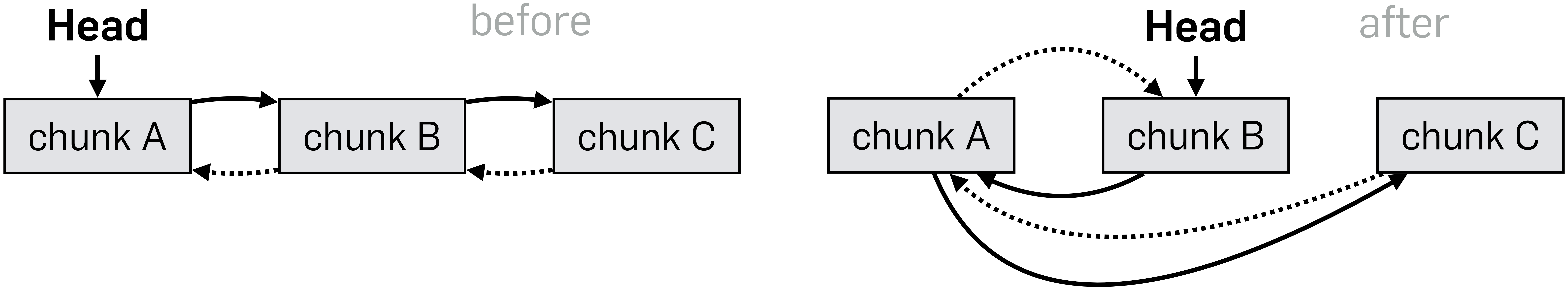
Finds:
Use after free (dangling pointers), heap buffer overflow, stack buffer overflow, global buffer overflow, use after return, use after scope, initialization order, memory leaks



Freelist Randomization

- Our list of free chunks so far was last-in-first-out (LIFO)
  - Fully deterministic and predictable
- If we randomize freelist regularly, we make it less predictable
  - Probabilistic defense though





Other Sanitizers

- AddressSanitizer to detect memory corruption
- MemorySanitizer to detect uninitialized memory
- ThreadSanitizer to detect race conditions and deadlocks
- UBSan to detect undefined behavior
- They all introduce (substantial) overhead and are rarely usable "in production"

Other Heaps

- Many other allocators and other types of allocators exist
  - Region/arena allocators
  - Bump allocators
- For example
  - jemalloc
  - Chrome's v8 heap allocator
  - XNU kernel and iOS userspace allocators
  - Linux kernel allocators

jemalloc

- Started in FreeBSD as a high-performance multi-processor DMA
- Used in Mozilla Firefox, Android, Redis, MariaDB, etc.
- Various flavors of jemalloc exist, slightly different
  - FreeBSD, Mozilla Firefox, Android, etc.
- Core goal
  - Performance, not security
- Core idea
  - Use multiple arenas at the same time for multi-processor performance
  - Reduce memory fragmentation

jemalloc

- Multiple arenas
  - A single thread may use
    - The same fixed assigned arena (e.g., via thread ID hashing)
    - Or a different arena with each malloc()
      - Different approaches: Pseudo random, round robin, timing, etc.
- Minimal memory page utilization is not crucial
  - Low fragmentation is more important

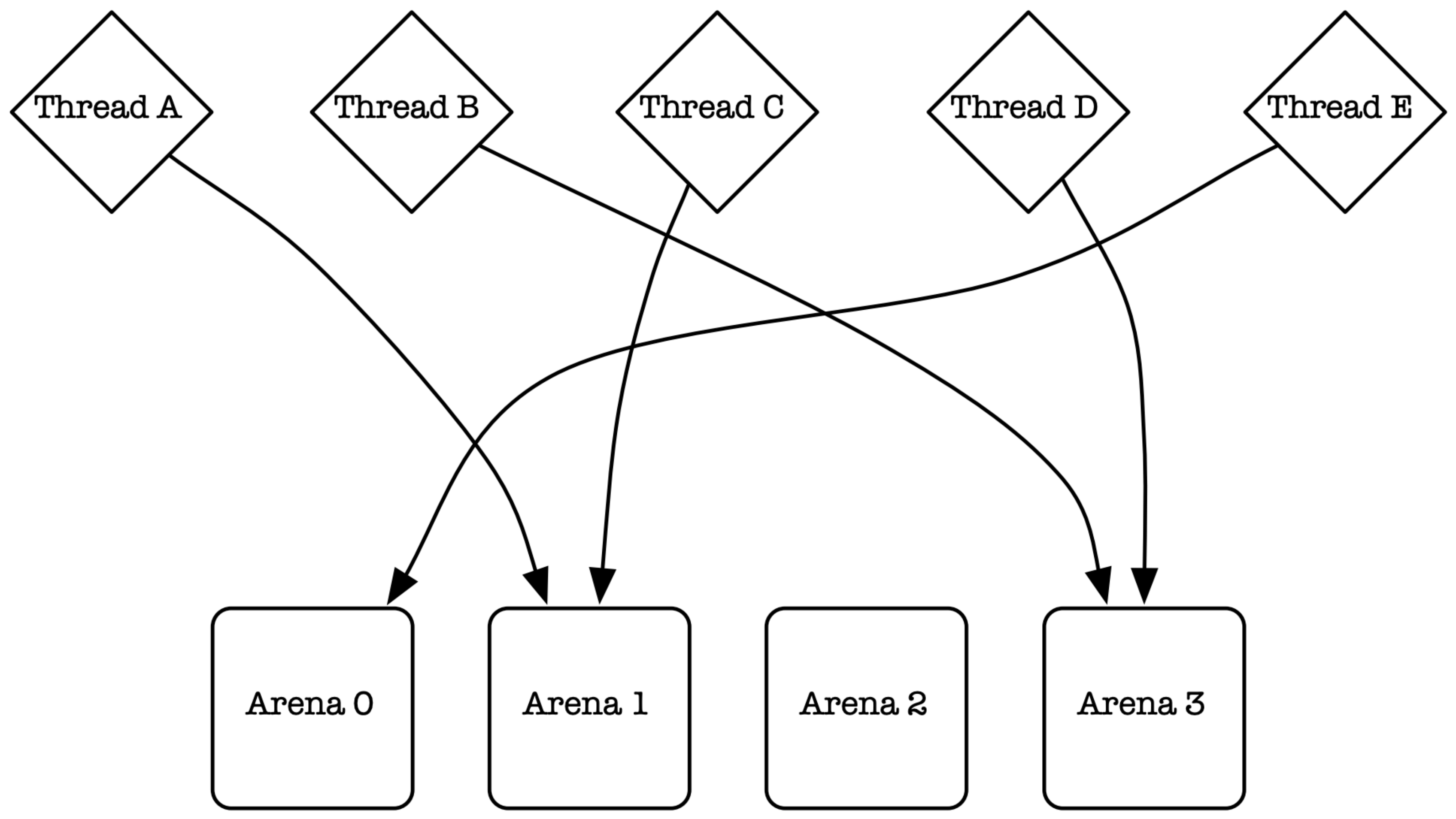


Figure via Jason Evans

jemalloc Terminology

- Chunk
  - 1024 contiguous 4K pages (=4MiB), aligned at 4MiB boundary
- Run
  - 1+ contiguous pages within a chunk
  - Only one allocation size
- Region
  - Contiguous bytes that can be used for (small) allocations

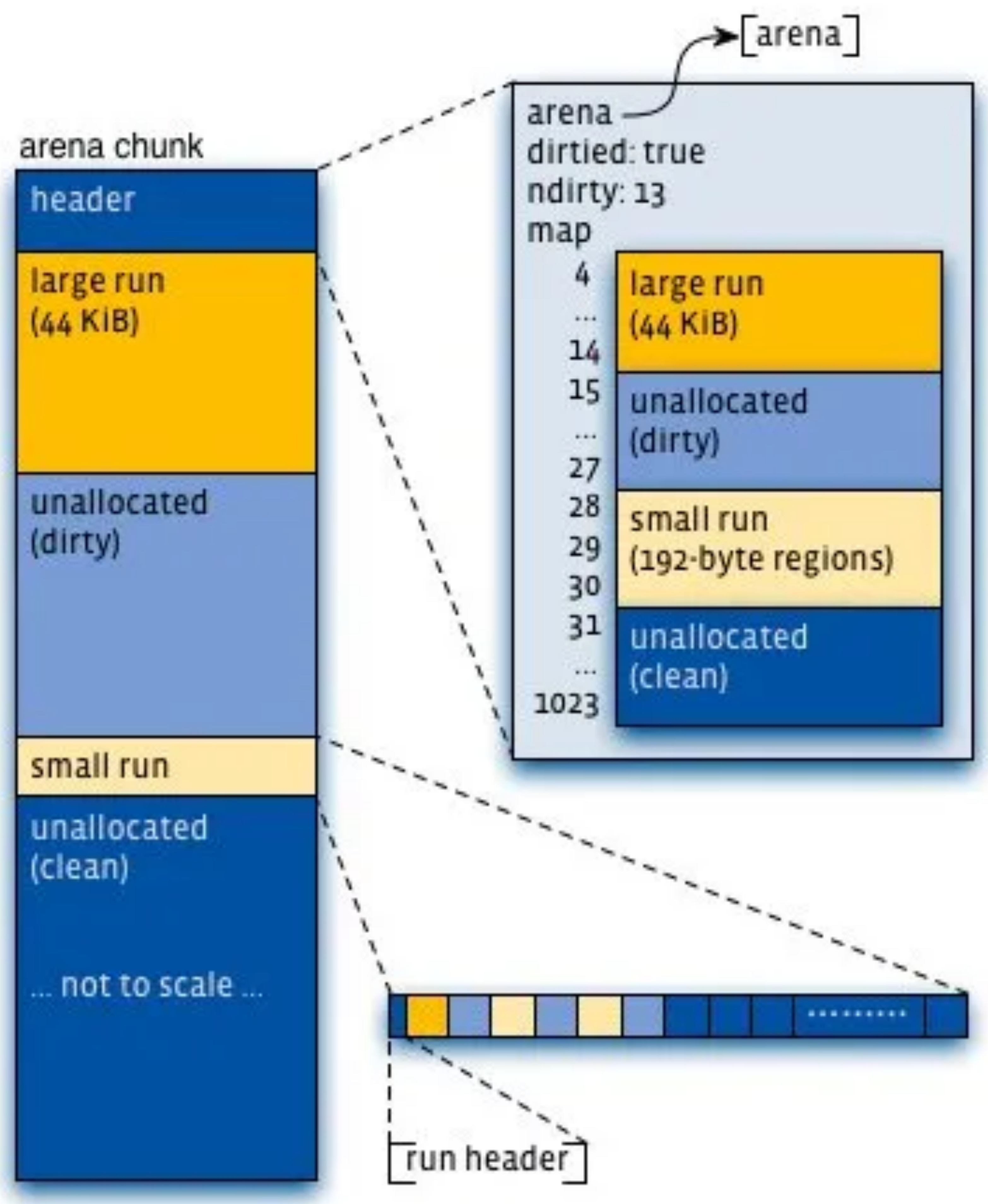


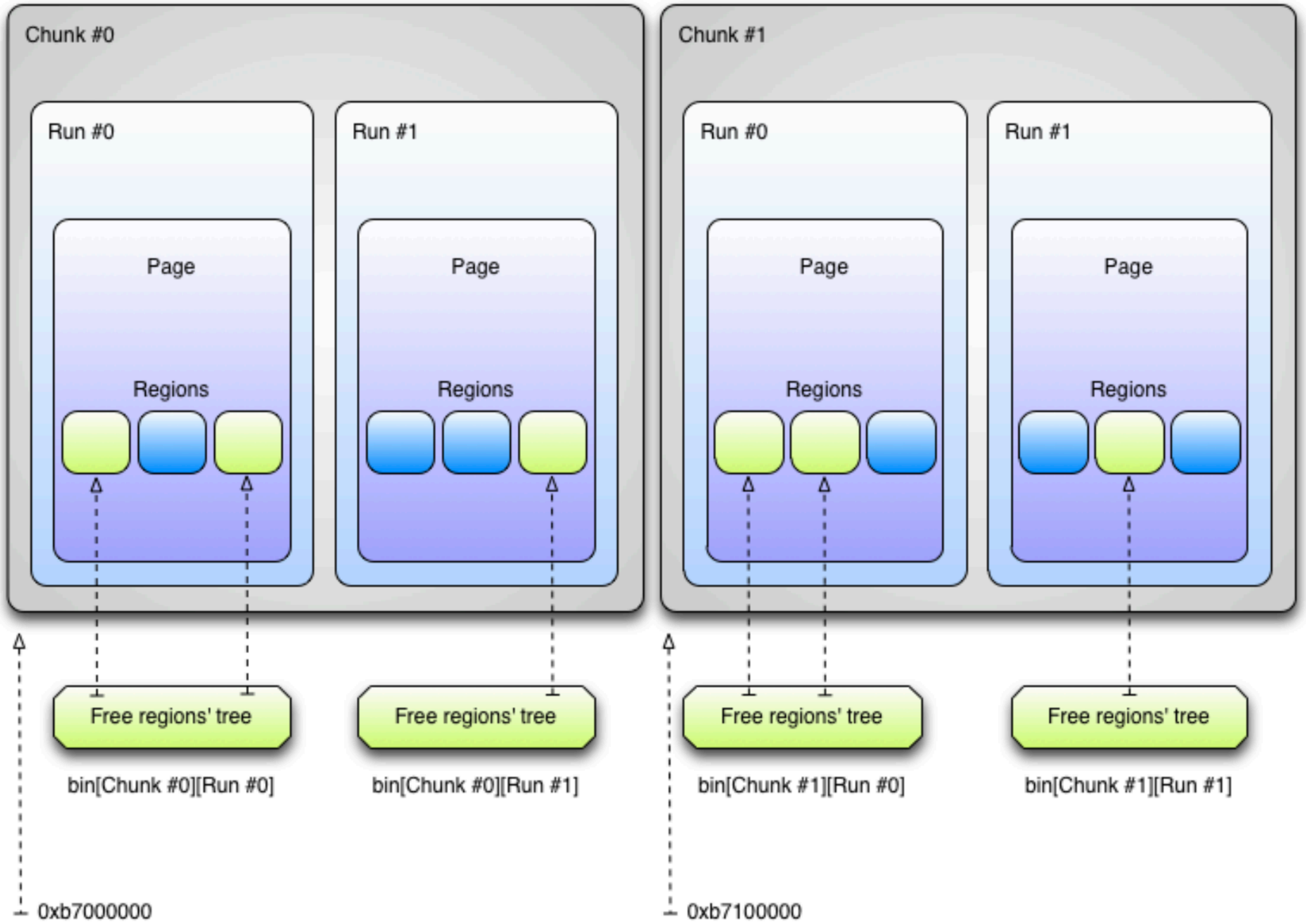
Figure via https://engineering.fb.com/2011/01/03/core-infra/scalable-memory-allocation-using-jemalloc/

jemalloc Allocation Sizes

Category	Subcategory	Size	
Small	Tiny	2	B
		4	B
		8	B
	Quantum-spaced	16	B
		32	B
		48	B
		...	
		480	B
		496	B
		512	B
	Sub-page	1	kB
		2	kB
Large		4	kB
		8	kB
		16	kB
		...	
		256	kB
		512	kB
		1	MB
Huge		2	MB
		4	MB
		6	MB
		...	

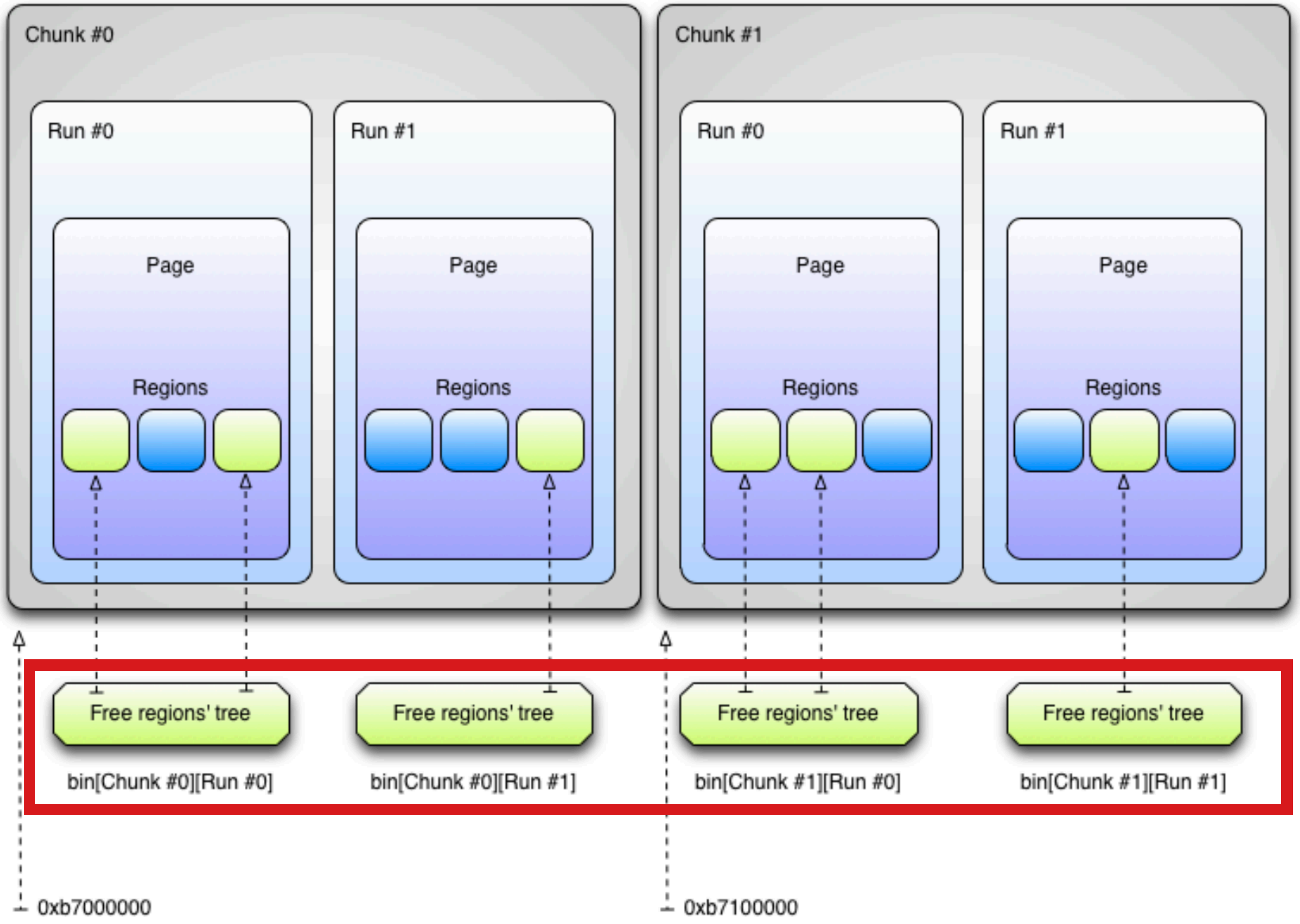
- Allocations always have specific sizes
  - Rounded up if needed
- Quantum-spaced allocs are needed
  - Most applications allocate small areas
  - If we only do 2^n, many allocations would waste memory ($\Rightarrow$ internal fragmentation)
  - This increases external fragmentation though ($\Rightarrow$ more runs to cover all sizes)

jemalloc Chunks, Runs, and Pages



via "Exploiting the jemalloc Memory Allocator: Owning Firefox's Heap," argp & huku

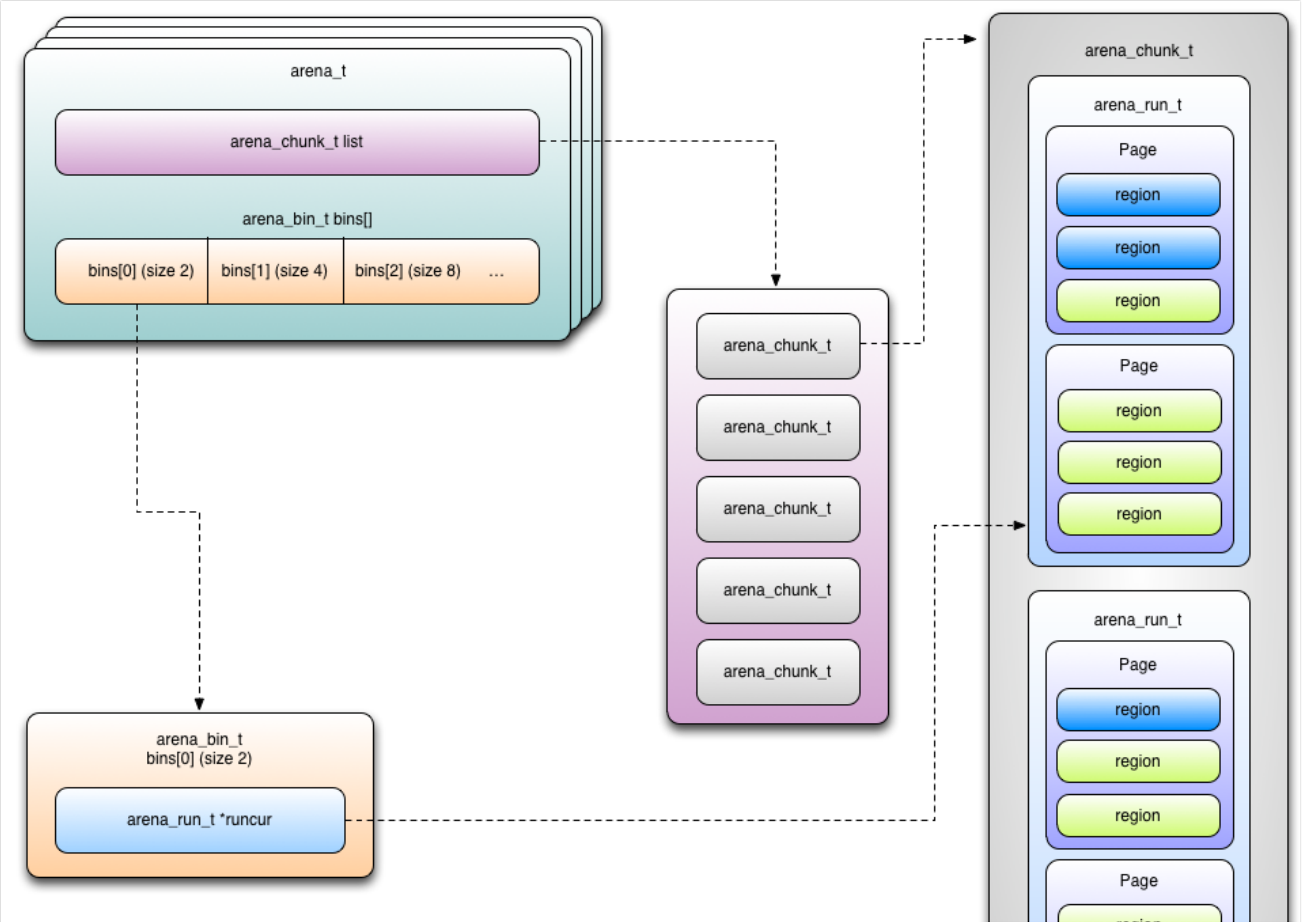
jemalloc Chunks, Runs, and Pages



Stored separately from actual chunk

via "Exploiting the jemalloc Memory Allocator: Owning Firefox's Heap," argp & huku

jemalloc High-level Overview

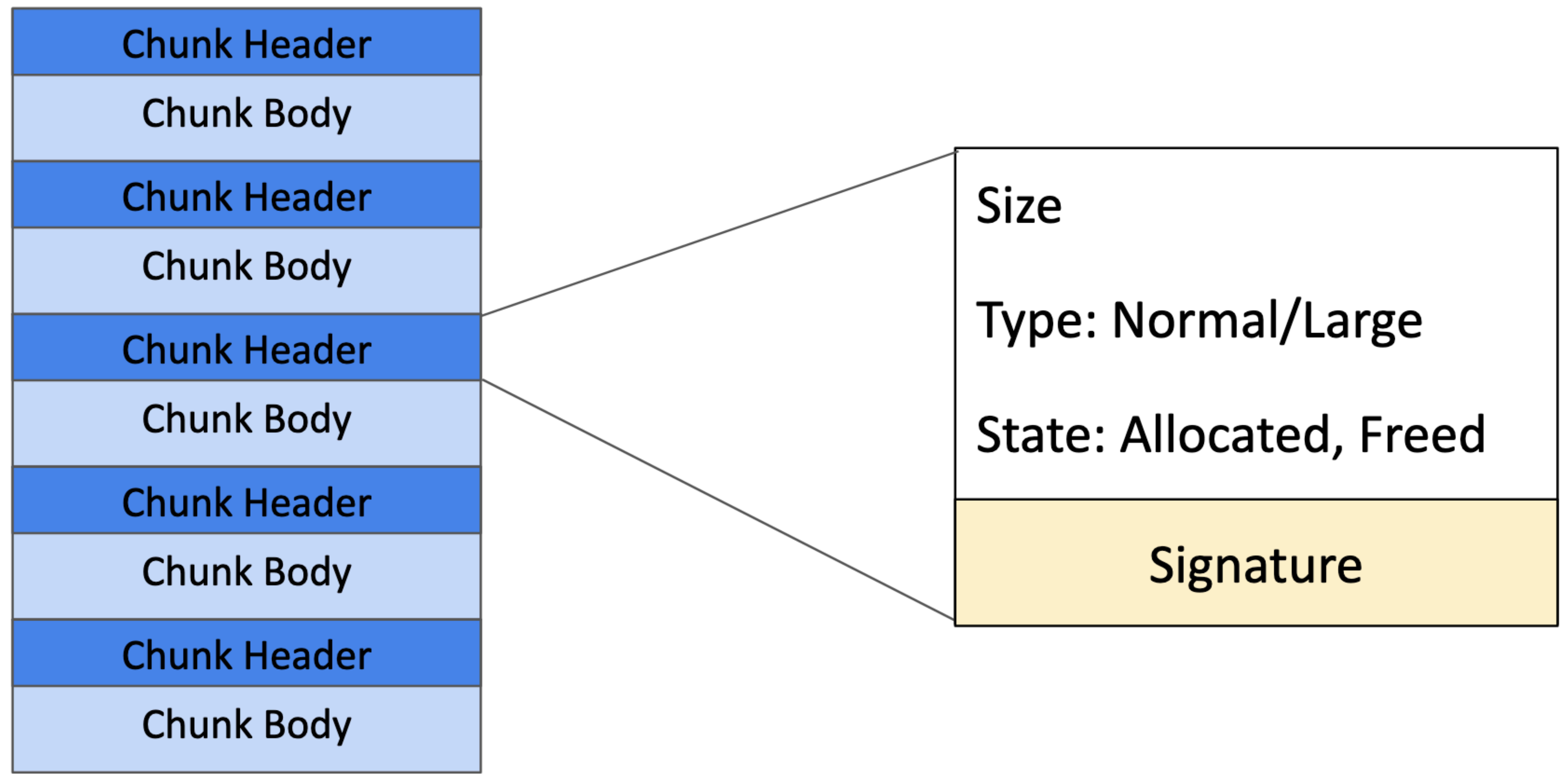


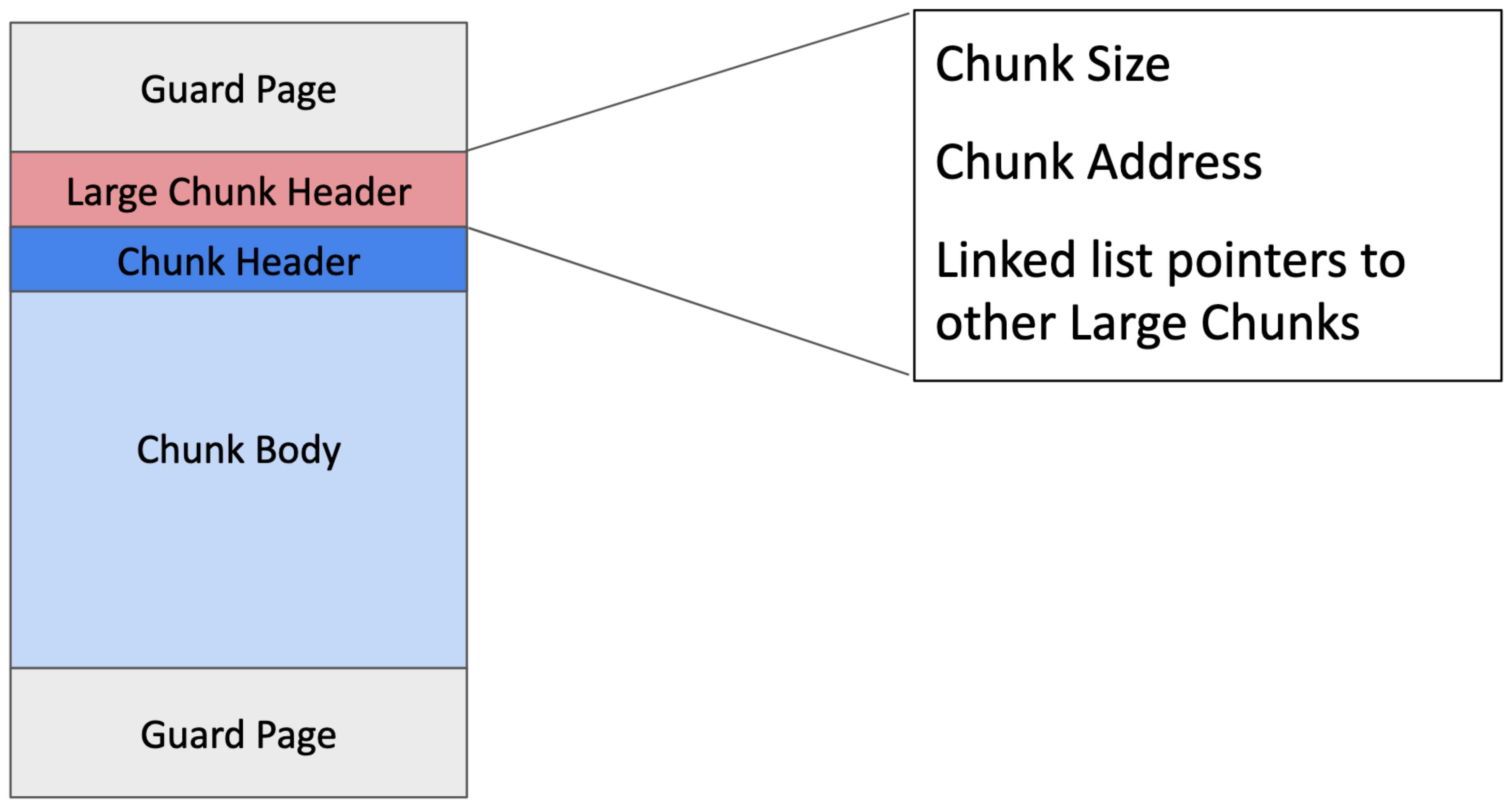
jemalloc

- By separating code and data, our only options are
  - Adjacent region overwrites
  - Corrupt run headers
  - Corrupt chunk headers
    - Remember chunks are different than for glibc

Scudo

- scudo is Android new hardened DMA
- Randomizes the address of allocations
- Signs chunk headers and verifies the signature before parsing the metadata
- Corruptible data
  - For normal chunks, only chunk header is inline
  - The large chunk header stores unprotected inline pointers
  - ⇒ Not a lot for us to work with





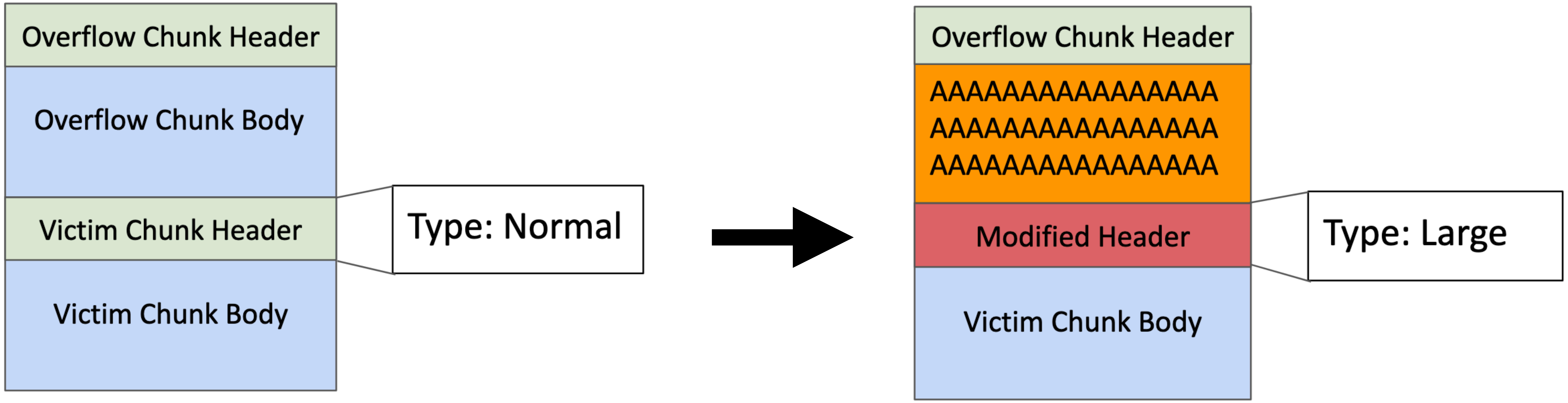
via "Exploiting Android's Hardened Memory Allocator," Mao et al.

Scudo

- Luckily, there are performance optimizations
- On fork(), we do not reinitialize
  - The randomization seed
  - Signing secret key
- This means
  - We can know the addresses of all chunks
  - We can manipulate the chunk header (and sign it)

via "Exploiting Android's Hardened Memory Allocator," Mao et al.

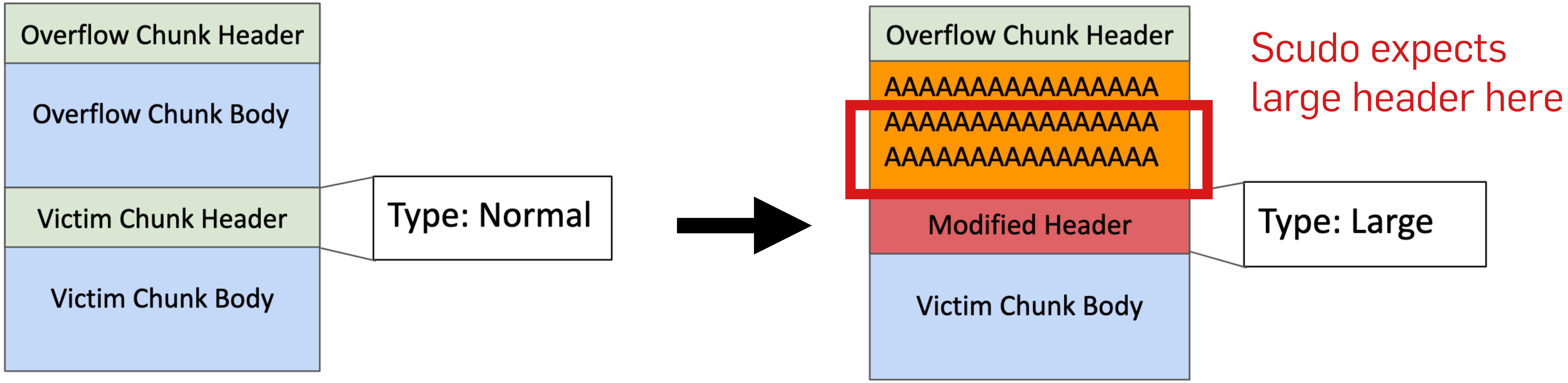
Scudo



- We arrange the chunk so the layout becomes exploitable
  - Doable because we can predict addresses due to randomization key
- We overflowing and modify the victim chunk (to large)
  - Doable because we can sign the modified header
- When we free victim chunk, then Scudo parses our attacker-controlled pointers

via "Exploiting Android's Hardened Memory Allocator," Mao et al.

Scudo



- We arrange the chunk so the layout becomes exploitable
  - Doable because we can predict addresses due to randomization key
- We overflowing and modify the victim chunk (to large)
  - Doable because we can sign the modified header
- When we free victim chunk, then Scudo parses our attacker-controlled pointers

via "Exploiting Android's Hardened Memory Allocator," Mao et al.

Scudo

- Fork() behavior enables two attacks
  - Forged commitbase
    - Directly insert a fake large chunk into the large chunk free list
  - Unsafe unlink
    - Modify linked list pointers to corrupt a normal chunk free list
    - Similar to unsafe_unlink for glibc 2.38
      https://github.com/shellphish/how2heap/blob/master/glibc_2.38/unsafe_unlink.c

via "Exploiting Android's Hardened Memory Allocator," Mao et al.

Type-based DMA / Type Isolation

- Yet another variant of arena/region allocators
- General idea: One arena/region per type
  - struct type_a is in arena 1, struct type_b is in arena 2, etc.
- Extremely efficient at eliminating type confusion attacks
  - Many/most modern attacks require type confusion vulnerabilities
  - If a struct of a specific type is used but its pointer does not match its type, fail at runtime/throw an exception
- In practice, it is not always possible to separate different types in different arenas (e.g., class inheritance)

Software Security 1

C++

Kevin Borgolte kevin.borgolte@rub.de

C vs. C++

"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off." (Bjarne Stroustrup, Inventor C++)

C++

- Started as "C with classes"
  - Not true (anymore), entirely different language
  - Has its own standards and guidelines
    - Now a modern programming language
  - Third most popular language after Python and C
  - Differs in undefined behavior (UB) to C
    - You can compile (most) C code with a C++ compiler, but your program might behave completely differently and might even have undefined behavior!

```c
#include <limits.h>
#include <stdio.h>

void main(void) {
  char *s = "foobar";
  s[0] = 'F';
}
```

OK in C
UB in C++

C++ Classes

```cpp
class M {
    std::size_t C;
    std::vector<int> data;

public:
    M(std::size_t R, std::size_t C) : C(C), data(R*C) {
        // constructor definition
    }

    int operator()(std::size_t r, std::size_t c) const {
        // member function definition
        return data[r * C + c];
    }

    int& operator()(std::size_t r, std::size_t c) {
        // another member function definition
        return data[r * C + c];
    }
};
```

- C++ supports object-oriented programming (not required like Java)
- Often used in C++, and how internals are implemented can lead to (security) issues

Memory Management

- Stack like in C
  - Stack variables cease to exist when they go out of scope
    - Scope can be smaller than function
- Dynamic memory management
  - malloc/free
    - Calling the C library, same behavior
    - **Strongly** discouraged
  - new/delete
    - Calls the constructor/destructor
  - Generally discouraged to be called directly in modern C++ (libraries should do it for you)

```cpp
#include <cstdlib>
#include <iostream>

class A {
  public:
    A() {
       std::cout << "A()" << std::endl;
    }
};

int main() {
   A* a = (A*)malloc(sizeof(A));
   std::cout << "malloc()" << std::endl;
   A* b = new A;
   std::cout << "new" << std::endl;
}
```

Type Casting

C

```c
#include <stdlib.h>

struct A {};
struct B {};

int main(void) {
  struct A *a = (struct A*)malloc(sizeof(struct A));
  struct B *b = (struct B*)a;
}
```

C++

```cpp
class A {};
class B {};

void cpp(void) {
  A *a = new A;
  B *b = (B*)a;
}
```

Type Casting

C

```c
#include <stdlib.h>

struct A {};
struct B {};

int main(void) {
  struct A *a = (struct A*)malloc(sizeof(struct A));
  struct B *b = (struct B*)a;
}
```

C++

```cpp
class A {};
class B {};

void cpp(void) {
  A *a = new A;
  B *b = (B*)a;
}
```

Compiles, but this is not the same operation as casting it in C

Type Casting

```c
#include <stdlib.h>                                           C

struct A {};
struct B {};

int main(void) {
  struct A *a = (struct A*)malloc(sizeof(struct A));
  struct B *b = (struct B*)a;
}
```

```cpp
class A {};  C++
class B {};

void cpp(void) {
  A *a = new A;
  B *b = (B*)a;
}
```

```cpp
class A {};                      C++
class B {};

void cpp(void) {
  A *a = new A;
  B *b = static_cast<B*>(a);
}
```

Compiles, but this is not the same operation as casting it in C

```
inherit.cpp:6:10: error: static_cast from 'A *' to 'B *',
which are not related by inheritance, is not allowed
  B *b = static_cast<B*>(a);
         ^~~~~~~~~~~~~~~~~~
1 error generated.
```

Type Casting in C++

- static_cast<T>(exp)
  - Compile-time cast using implicit and user-defined conversions
- dynamic_cast<T>(exp)
  - Runtime cast between classes, can require runtime type information (RTTI)
- const_cast<T>(exp)
  - Used to remove constness
  - Can you think of a reason/place where this might be useful?
- reinterpret_cast<T>(exp)
  - Compile-time directive, treat exp as type T

Run-time Type Information

- Effectively for every polymorphic class (with virtual functions), the compiler generates an instance of a new class and points to it in the virtual table of the original class
- The information can be useful for recovering class structures when reverse engineering (but might be obfuscated)

```
/* part of the standard library */
class std::type_info {
  /* implementation-specific information */
  private:
    const char *__type_name;
};

/* class B with base/parent class A */
class __B_type_info: public __A_type_info {
  public:
    const __A_type_info *__base_type;
};
```

Virtual Table

- A virtual function in C++ is a function that is being re-defined by a sub/derived class
- A purpose for them is runtime polymorphism/late binding
- The program determines at runtime which function to call through the virtual table (vtable)

```
# vtable for SubClass
0                         ; offset to base
typeinfo for SubClass     ; type info pointer
SubClass::vfunc1(void)    ; first virtual function
BaseClass::vfunc2(void)   ; second virtual function
```

Virtual Table: Example with Inheritance

```cpp
#include <cstdio>

struct Bar {
    virtual void method1() {
        std::puts("method1: this is bar");
    }

    virtual void method2() {
        std::puts("method2: this is bar");
    }
    virtual ~Bar() {};
};

struct Foo: public Bar {
    virtual void method1() {
        std::puts("method1: hi, I'm foo");
    }
    virtual ~Foo() {};
};

int main() {
    Bar bar;
    Foo foo;
    Bar* bar_arr[2];
    bar_arr[0] = &bar;
    bar_arr[1] = dynamic_cast<Bar*>(&foo);

    for (auto ptr: bar_arr) {
        ptr->method1();
        ptr->method2();
    }
    return 0;
}
```

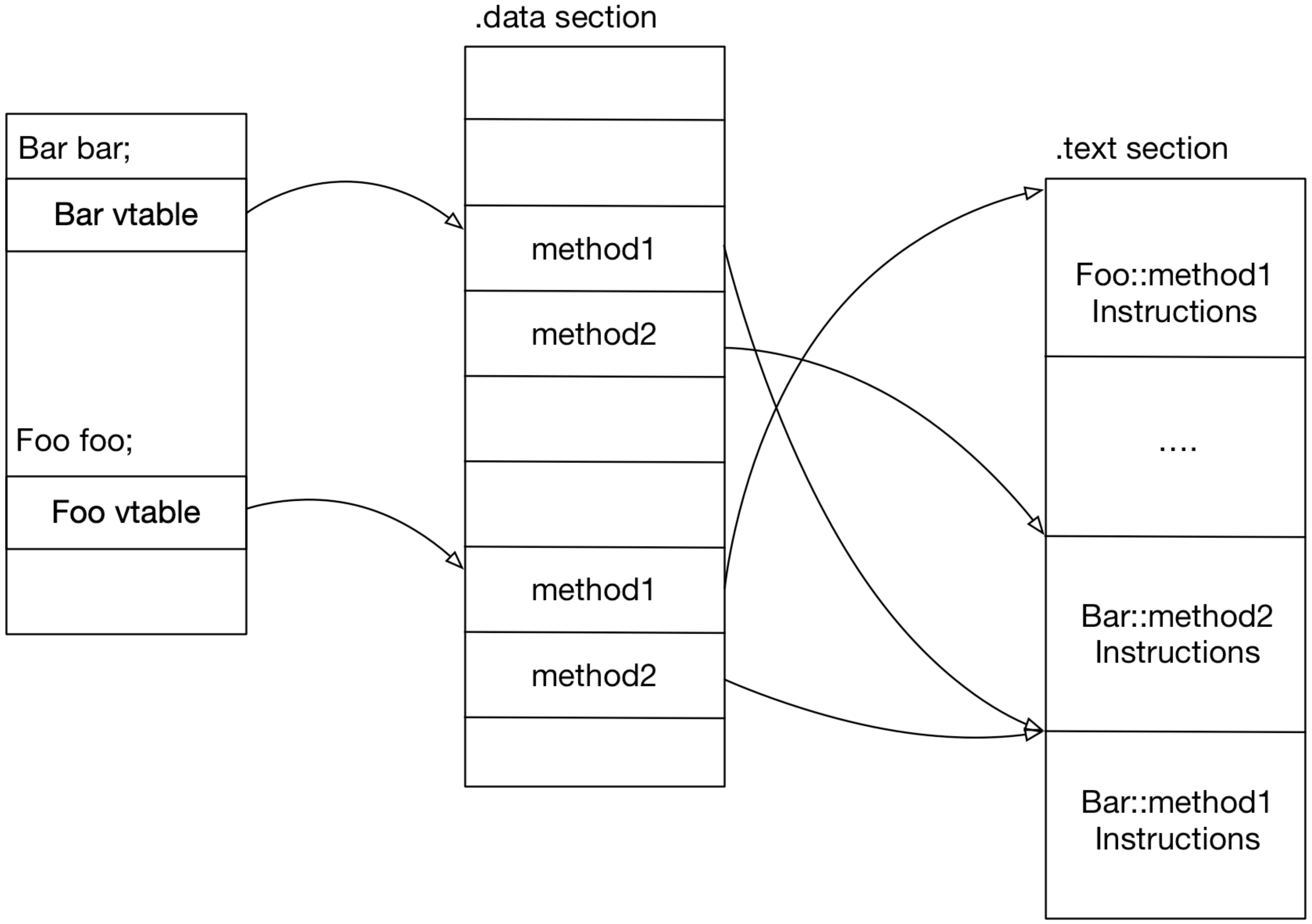


Figure via https://guihao-liang.github.io/2020/05/30/what-is-vtable-in-cpp

Virtual Table: Attacks

- Layout of vtable depends on the compiler

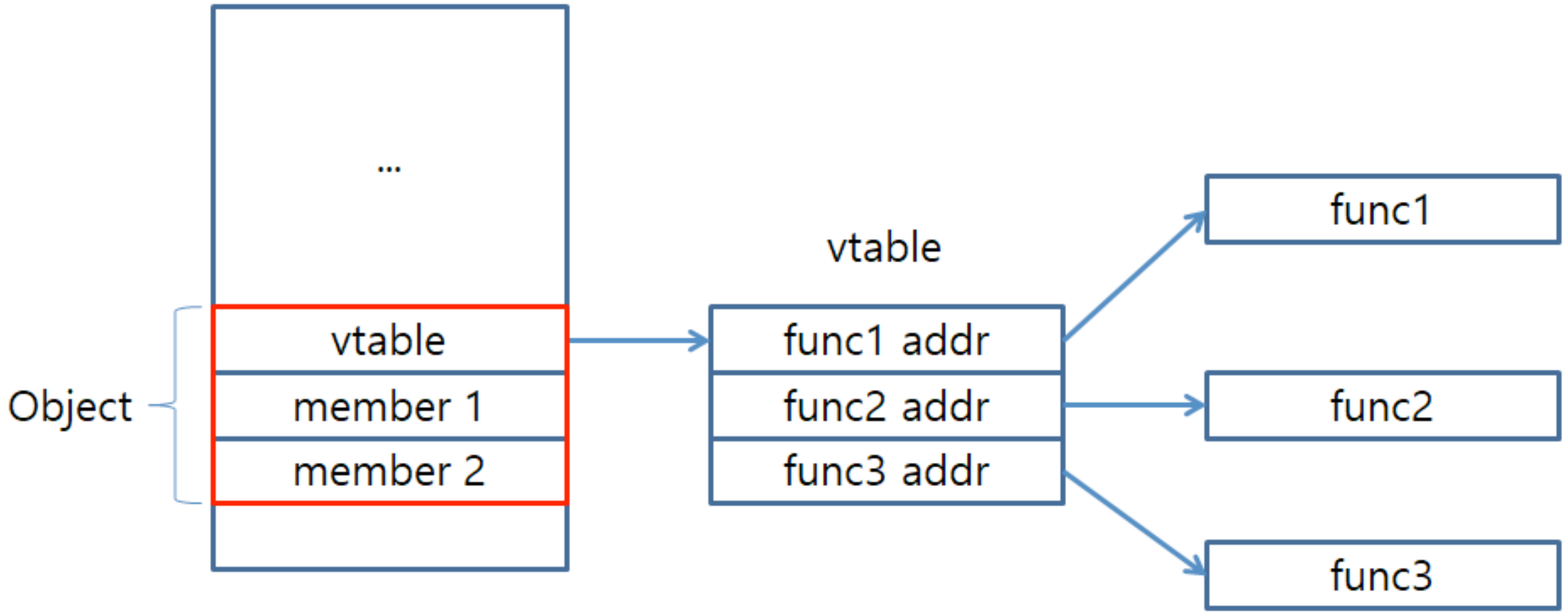


Figure via https://linarena.github.io/windows-0x0b

Virtual Table: Attacks

- Layout of vtable depends on the compiler
- Overwriting entries in vtable can be possible
  - This means we change where a virtual member function is

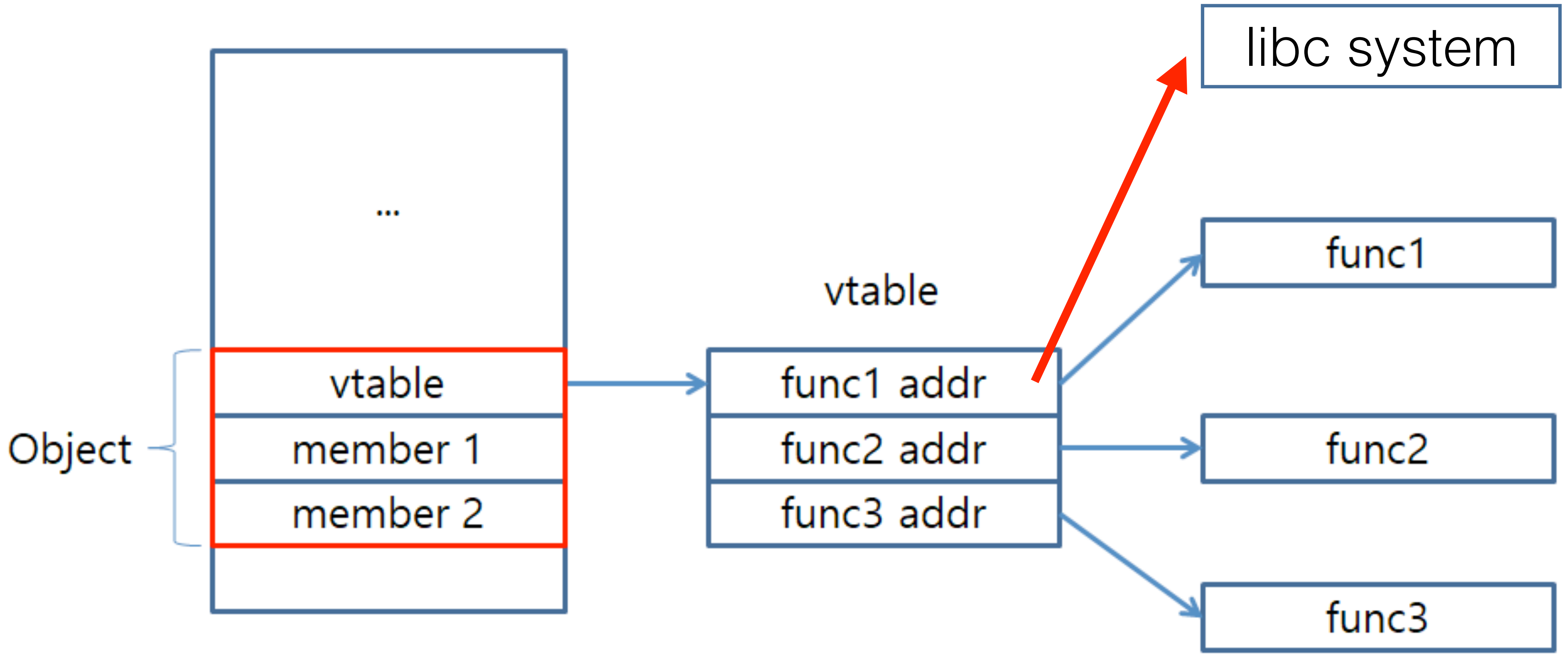


Figure via https://linarena.github.io/windows-0x0b

Virtual Table: Attacks

- Layout of vtable depends on the compiler
- Overwriting entries in vtable can be possible
  - This means we change where a virtual member function is
- Overwriting the pointer to the vtable possible
  - This means we change the type of the variable (we need to ensure that the vtables are somewhat compatible)

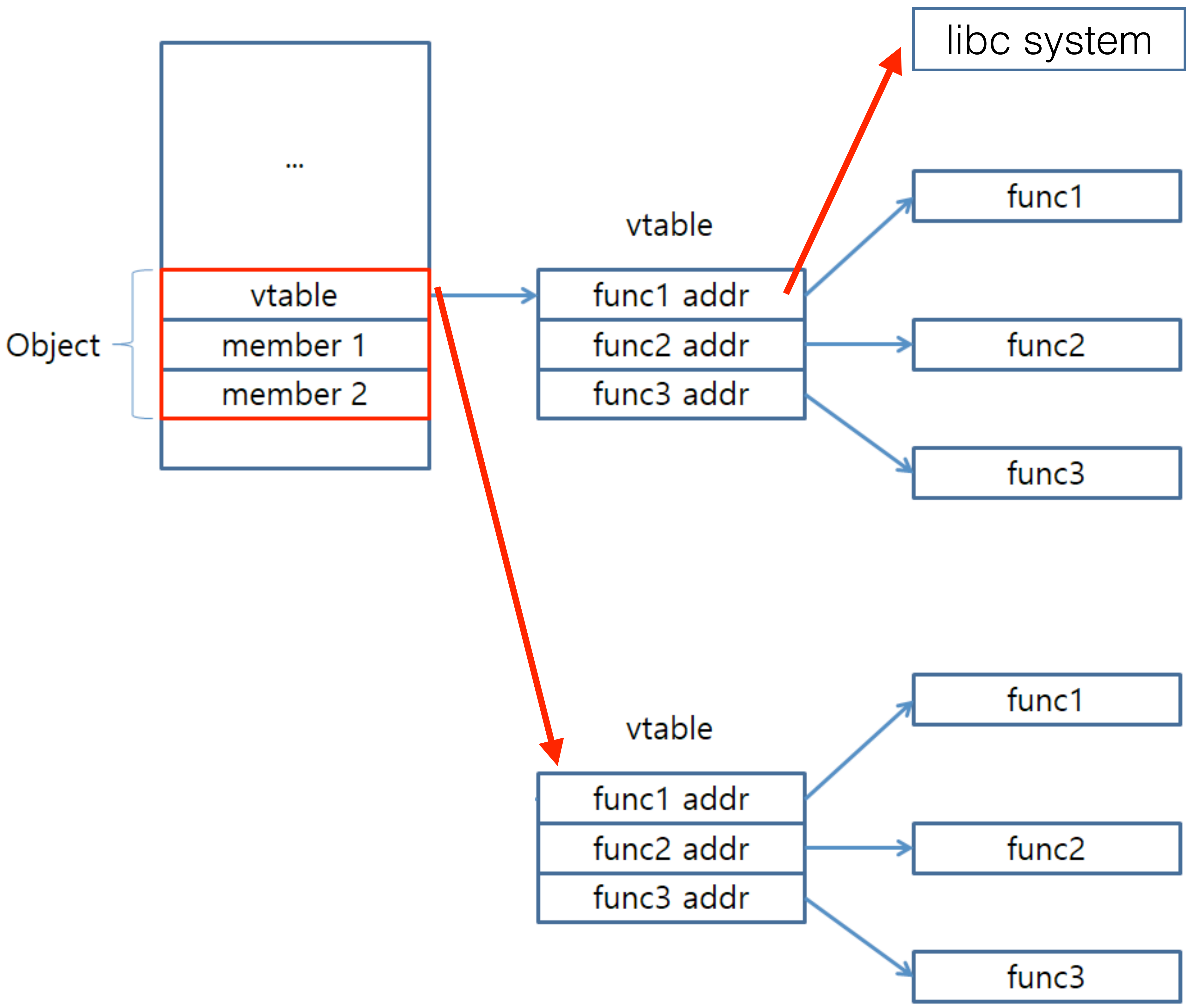


Figure via https://linarena.github.io/windows-0x0b

Virtual Table: Security

- Security protections for vtables exist
  - Virtual table verification (GCC), which limits the set of allowable virtual functions
    https://gcc.gnu.org/wiki/cauldron2012?action=AttachFile&do=get&target=cmtice.pdf
  - vtguard (Microsoft Visual C++), basically a stack canary for the virtual table
    https://media.blackhat.com/bh-us-12/Briefings/M_Miller/BH_US_12_Miller_Exploit_Mitigation_Slides.pdf
  - Heap partitioning
- See also "Smashing C++ VPTRs" by rix (http://phrack.org/issues/56/8.html)

Smart Pointers

- Pointers can be dangerous
  - Plus they are difficult to analyze statically
- Dynamic memory management is difficult to do properly
  - Which function allocates?
  - Which function frees/deleted?
  - Has it been free'd/deleted?
- Modern C++ and other languages have smart pointers

Smart Pointers

- Smart pointers keep track of memory for you (limited garbage collection)
  - If all smart pointers to an object go out of scope, it is automatically delete'd
- Concept of ownership
- C++ supports
  - shared_ptr
    - Non-exclusive ownership of an object (will be delete'd when all go out of scope)
  - unique_ptr
    - Takes exclusive ownership of an object
  - weak_ptr
    - Temporary ownership, a reference that is not owned and might change