A Security Analysis of CheriBSD and Morello Linux

Dariy Guzairov[1][0009-0001-5287-0556], Alex Potanin[1][0000-0002-4242-2725], Stephen Kell[2][0000-0002-2702-5983], and Alwen Tiu[1][0000-0002-2695-5636]

[1] Australian National University, Australia
{dariy.guzairov,alex.potanin,alwen.tiu}@anu.edu.au
[2] King's College London, United Kingdom
stephen.kell@kcl.ac.uk

Abstract. Memory corruption attacks have been prevalent in software for a long time. Many mitigation strategies against these attacks do exist [22], but they are not as far-reaching or as efficient as the CHERI architectural extensions [27]. CHERI uses capabilities to restrict pointers to certain regions of memory and with certain access restrictions. These capabilities are also used to implement "compartmentalisation": dividing a binary into smaller components adhering to the principle of least privilege. At present the compartmentalisation mechanisms are effective in containing malicious code to its compartment only in limited circumstances. This paper details four ways to bypass compartmentalisation, with a focus on CHERI ports of the Linux and BSD operating systems. We find that simple bugs and attacks can still allow malicious code to bypass compartmentalisation if certain powerful APIs are available within the process; while these attack surfaces are formally outside the attack model, developers may not realise that these surfaces are exposed. We conclude with mitigation measures to prevent these attacks, a proof-of-concept demonstrating their use, and recommendations for further securing compartmentalisation within Linux and BSD.

Keywords: CHERI · Capability · Compartmentalisation · Security Analysis

1 Introduction

Hardware-enforced security mitigations are a popular research topic [29, 27]. CHERI architectures appear especially promising, and have been promoted by the UK government as the "new standards in secure system design" [23]. Their core feature is *capabilities* augmenting pointers with bounds and permissions on memory—in effect a hardware implementation of "fat pointers" [17]. Each stored capability typically occupies 129 bits: a 64-bit address plus bounds and permissions, and a disjoint validity bit providing unforgeability. These features were added to AArch64 in the Morello prototype silicon, on which we conducted this paper's experiments. The design intention is that code written in stock C

and C++ can be compiled for CHERI architectures and thereby gain memory safety guarantees, comparable to those of Rust or other more invasive software approaches with relatively small overheads of porting or performance [28].

Compartmentalisation is a further concept of CHERI architectures, with the aim of finer-grained separation within individual process, thereby separating *untrusted* from *trusted* code and data. This is achieved through use of special opaque or 'sealed' CHERI capabilities, which can be validly exchanged without allowing libraries to access each other's code and data. In this paper we show that, while CHERI architectures provide easily adopted features for preventing memory corruption bugs, there remain limitations when it comes to realising secure compartmentalisation of real-world software systems and libraries. We discuss two platforms, Morello Linux and CheriBSD, and show how malicious code can bypass compartmentalisation, leading to a final proof-of-concept in which a malicious library retrieves a private key from the main binary.

We use the same threat model as Gutstein [9]: an *attacker that has arbitrary code execution within an unprivileged library of some process and aims to gain code execution or access to secrets in a more privileged library.* Such an attack scenario is plausible in the case where a malicious library is used within an application, e.g., through a supply chain compromise [5, 14]. We consider only attacks that compromise memory safety in user space, i.e. the malicous code is not assumed to run with system privilege. We also exclude from scope attacks that make direct system calls (such as calling `execve()`) or access (pseudo-)file systems (such as `/proc/self/mem` on Linux). The malicious code can, however, call functions in common libraries such as `libc`. We show how such a malicious library is able to break compartmentalisation and access secrets in a more privileged compartment, demonstrated by reading or writing to a variable in the main binary, such as a private key or a flag.

We assume CHERI's hardware primitives are secure, so the only way to obtain capabilities is to manipulate existing ones. Our exploration thus focuses on examining ways in which capabilities can be (unintentionally) leaked in the presence of compartmentalisation. Specifically, our contributions are as follows.

- *Dynamic linker exploit.* We show that in both Morello Linux and CheriBSD, library calls that interface with the dynamic linker (such as `dlopen()`) allow malicious code to obtain capabilities to all loaded libraries and the main binary.
- *Memory scavenging.* We show that capabilities are inadvertently left on the stack or heap objects, e.g., when a function does not zero a buffer prior to exiting or the heap allocator does not zero freed objects. These can be accessed by the malicious code through systematic scavenging, potentially allowing an attacker to obtain further capabilities through a recursive scan of the stack or heap.

Theoretical concerns about capability leakage have been discussed in [9, 11]. Gutstein [9] anticipated the threat model and the general danger of valid-capability misappropriation, while Hammond et al. [11] formalised conditions under which such leakage must not occur. Of note is that the `dlopen()` exploit

we found still lies within the trusted computing base (TCB) assumed in Gutstein model (i.e., the dynamic linker is considered part of the TCB in the threat model in [9]). Hammond et al. [11] shows an example of enforcing compartmentalisation under highly idealised conditions (e.g., no library calls at all involved in the compartmentalised code). Our results do not contradict these prior results – rather, our exploits expose the gap between the assumptions underlying these prior studies and the behaviour of deployed CHERI software stacks.

Proof-of-concept code for our exploits is available.[3]

Ethics and Disclosure This research discovers and demonstrates four attacks that bypass compartmentalisation in Morello Linux and CheriBSD. We followed responsible disclosure, notifying the CheriBSD and ARM (Morello Linux) development teams in mid to late 2025, with detailed descriptions of the vulnerabilities and the proof-of-concept code. The CheriBSD developer team has responded acknowledging a gap in the documentation surrounding the APIs that can defeat compartmentalisation, and that this will be improved to explicitly mention their limitations with respect to compartmentalisation. They also suggested better tooling could be implemented to identify API accesses in the compartmentalised code, preventing those API misuses in the first place.

Our experiments were conducted on an emulator (QEMU) and a prototype hardware (ARM Morello boards) in a controlled lab environment; no production systems or user data were targeted or compromised.

2 Background

2.1 CHERI Architectural Extensions

While CHERI architectures are extensions of conventional RISC-style architectures, capabilities drastically change what can be done. In current 64-bit CHERI architectures, a capability consists of a 64-bit address, a base, a length, an offset, a validity bit, and a permission mask [26]. The base, length, and offset describe the region of memory the capability can interact with, while the permission mask describes what the capability can do, such as reading or writing. The validity bit is not stored inline with the capability, as it dictates whether the capability is valid, preventing the forging of capabilities. Below is an example of different capabilities (presented in a "pretty printed" format):

```
0x1 0x0000fffffff7c9f0 [rwRW,0x0000fffffff7c000-0x0000fffffff80000]
0x1 0x0000000040195cc0 [rwRW,0x0000000040195cc0-0x0000000040195d20] (sealed)
0x1 0x0000000040152d61 [rxR, 0x0000000040130000-0x0000000040190700] (sentry)
0x0 0x0000000040c19000 [rwRW,0x0000000040c19000-0x0000000040c19010] (invalid)
```

We see a 64-bit memory address followed by permission bits that are (here) either read and write (`rwRW`), or read and execute (`rxRE`), and the address range the permission applies to. The lowercase permissions apply when reading and writing standard data, while the uppercase ones apply when reading and writing

[3] `https://github.com/alwentiu/cheri_security.git`

capabilities. For example, when loading through a capability lacking the uppercase read bit, any capabilities read will be invalidated.

The final line above shows an invalid capability (tag bit 0). All registers and 128-bit-aligned units of memory have a validity bit: 0 for plain data, 1 for capabilities. A valid bit is cleared either explicitly in code, when a capability is outside of the set range, or when non-capability data is read. The tag bit is stored separately from the 128-bit capability and cannot be addressed in code, so malicious code cannot craft a capability pointing to an address of its choosing.

As seen in the middle lines, there are also *sealed* and *sentry* capabilities. A sealed capability is immutable and cannot be dereferenced [25]; this is useful for protecting global data or data passed to an untrusted compartment. A sealed entry point ('s-entry') is an executable capability that cannot be used for loads or stores but can be called.

CHERI's primitives allow for a wide range of use cases. Capabilities can be fine-grained, with a narrow byte length, or larger, spanning many pages. The sealing primitive can also be used in many ways: capabilities are sealed with a sealing capability, and the original can only be unsealed when holding that same sealing capability. Attempting to dereference a sealed capability causes a trap. The sealing primitive is the foundation for compartmentalisation.

Morello Linux provides a *global sealing capability* through the API `getauxptr(AT_CHERI_SEAL_CAP)` for sealing private data. If an attacker obtains the global sealing capability, or any other capability used for sealing, they may be able to unseal and access it as usual. The Morello SDK documentation discusses such an exploit scenario and possible ways to mitigate it.[4] In our attack model, we do not assume that the attacker is able to directly access such a global sealing capability, but we will instead show that it can be obtained indirectly through an exploit technique.

2.2 Compartmentalisation

Compartmentalisation confines untrusted code by isolating software components into separate compartments, e.g. to prevent attacks on privileged code [26, 15]. Prior work has explored software-only techniques [12] and hardware-based approaches [13]. Multiple design patterns for compartmentalisation using CHERI primitives have been documented [28], e.g. *sandboxing*, *assured pipelines*, *horizontal compartmentalisation*, *library compartmentalisation*, and *temporal compartmentalisation* In all cases the aim is to prevent untrusted code from 'escaping' its compartment, by granting it a limited set of rights and encapsulating *trusted* components within their own compartments.

We focus on bypassing compartmentalisation within a single process, as intra-process isolation is CHERI's advantage over process-granularity isolation and inter-process communication (IPC). Each process brings greater overhead than

[4] See the Morello SDK repository `https://git.morello-project.org/morello/morello-sdk/`, under `morello/examples/compartments/`, for an example usage and discussions around its security implications.

a CHERI compartment and can share data only at page granularity, whereas CHERI allows fine-grained confinement within a single process via capabilities restricting access to subsets of the address space [28]. Owing to the shared address space, there is no overhead when sharing data among compartments at any granularity. However, malicious code may break out of its compartment via a leak of one or more capabilities.

2.3 The Dynamic Linker

Programs can be linked statically or dynamically. Dynamic linking avoids bundling every library with the binary; instead, the dynamic linker loads required libraries at run time. This can save space in memory and on disk, facilitate software updates, and so on.

Most dynamic linkage schemes involve a *global offset table* (GOT) in the memory of each binary (library or executable). Data and code that may be external to that binary are addressed indirectly through that binary's GOT. Code for calling a library function indirects through the GOT, which is often filled 'lazily' and so initially contains a placeholder address pointing to the dynamic linker. When this placeholder is called, the dynamic linker substitutes the real function address into the GOT, then calls the requested function. To perform such operations, the dynamic linker must keep metadata about each loaded binary, such as base address, name and symbol table. As such, the dynamic linker has extensive permissions and access to the binary, including on CHERI. As dynamic linking is often the default, the dynamic linker is ubiquitous and makes an interesting target, especially in the context of compartmentalisation.

2.4 System Setup and Configuration

Our experiments were conducted on the ARM Morello prototype board, an implementation of the CHERI-extended ARMv8-A architecture. We used two primary operating systems: Morello Linux (based on the musl C library) and CheriBSD (based on FreeBSD).

For both systems, code was compiled with the CHERI-augmented LLVM toolchain. The default compiler and runtime configuration significantly impacts the feasibility of certain attacks. For example, while CHERI-LLVM can produce bounded stack pointer capabilities to prevent stack walking across compartment boundaries, our Morello Linux experiments disabled stack-pointer bounding (we omitted `-fcheri-stack-protection`), so the dynamically-linked malicious library received the ordinary, unbounded stack-pointer capability, spanning previous frames. In CheriBSD, we tested both the base system (purecap dynamic linking only — neither sealing nor `c18n`) and the experimental `c18n` library-based compartmentalisation framework, which explicitly addresses these concerns.

3 Related Work

The CHERI architecture and its implications for software security have been extensively studied. Watson et al. provided the foundational overview of CHERI's hybrid capability model and its potential for scalable compartmentalisation [28]. Recent work has focused on formalising these guarantees; for instance, the Morello-Cerise project established a proof of strong encapsulation for the ARM Morello architecture [11].

Specific to compartmentalisation, the `c18n` framework in CheriBSD is a suite of experimental library-based isolation features implemented mainly within the dynamic linker [8]. Elsewhere, the CHERIoT real-time operating system (RTOS) shows how CHERI can enable fine-grained compartments even on low-cost embedded devices, using a specialised allocator and compartmentalisation-aware scheduler for strong isolation [1].

Stack protection and management in capability-based systems have also been studied [20], where formal reasoning for local capabilities to support safe stack and return pointer management is proposed, to provide a model for how a machine can provably prevent stack-based leaks. Our work adds to this literature by demonstrating the practical gaps in current prototype OS implementations (especially Morello Linux) when these formal properties are not fully realised or are curtailed for reasons of performance or compatibility.

Temporal safety and capability revocation are known challenges in CHERI. While the ISA provides primitives for spatial safety, temporal safety (e.g., preventing heap scavenging or use-after-free) requires co-design with the memory allocator. Prior work by Gutstein highlighted the security analysis of these mechanisms, noting that secure revocation is not guaranteed by hardware alone but must be supported by an appropriate allocator [9]. Our research provides empirical evidence of these limitations in current prototypes.

3.1 Software exploitation

As our attacks are relatively novel, they do not rely on much of the existing exploitation background. The most notable existing attacks and methodologies are *egghunters* and *sandbox escapes*.

Sandboxes aim to isolate untrusted code, limiting what it can do, e.g. reading or writing only a small subset of files or calling only a subset of functions [18]. For example, in a web browser, sandboxes are often child processes with restrictions such as only rendering a page and running JavaScript.

While stack and heap exploitation may give malicious actors code execution within a sandbox, sandbox escapes bypass the sandbox itself [4]. These escapes often rely on overflows or other logic bugs. In our case, compartmentalisation plays a similar role, confining malicious code to a small memory region. CHERI means we cannot use buffer overflows or similar memory corruption attacks to escape, but must resort to new techniques.

Unlike sandbox escapes and stack/heap exploitation, egg hunters search memory for a specific byte sequence [19]. Specifically, they firstly use a system call

to probe for mapped memory (unmapped addresses will return `-EFAULT`), then secondly test whether it contains a useful byte sequence. In this way, a small piece of code can find and execute a larger payload. Our use of egg hunting is distinct from prior uses: the interesting byte sequences we seek include valid capabilities and private keys.

4 Threat Model

We consider an attacker with arbitrary code execution within an unprivileged library or piece of code [9]. Specifically, we follow the model of Gutstein: "a controlled library that does not contain implementations of the TCB, such as the heap allocator or the dynamic linker, ... [and] can only access particular global functions to which it has been linked". Thus, the malicious library may call common C library functions, such as `malloc` or `dlopen`, for one of our attacks. Since we do not implement these functions, only call them, we stay within the threat model. System calls such as `execve` are also out of scope.

We believe this threat model captures a likely scenario for a malicious actor gaining code execution on CHERI, especially given recent high-profile supply chain attacks [5]. We assume the attacker wants to access data in another library or in the main binary that they currently cannot reach, due to compartmentalisation and capability restrictions. Our goal is thus to read or modify data held by the trusted binary but inaccessible to the malicious code. We expect the malicious library to be compartmentalised, with a distinct executable code region, global offset table, and stack. This is the key security feature we aim to break.

For example, in Hammond et al. [11], there is a variable in memory that only trusted code may read or write. This is the target of our malicious code when testing our vulnerabilities against this or similarly-designed code. A similar target exists in the example code by Brodsky et al. [3], where a key is sealed by a secure sealer capability. The key is global, but the sealing capability is only accessible to trusted code. Our aim is to obtain the sealing capability and then unseal the data. Our proof-of-concept attack has a similar goal: a private key generated by OpenSSL that is not accessible to the malicious library.

5 Recursive Capability Scanning

Before discussing our attacks, we introduce a technique that proved useful across all of them and shows how an attacker can leverage capabilities to break compartmentalisation. Since CHERI requires pointers to be derived from existing capabilities, malicious code cannot create arbitrary pointers or modify them to arbitrary addresses. To determine reachable memory, we use a recursive capability scan. Given a valid capability, this yields a list of capabilities accessible from it, enabling us to determine whether we have broken compartmentalisation, and letting the malicious library read and write any of these capabilities.

Since a capability specifies both the memory region it can access and its length, we read from the start to the end address; whenever we find a valid

capability, we scan it recursively. We also track capabilities and regions already seen: if new, we add it to a list and recurse; if already seen, we skip it. In essence, we compute the transitive closure of the initial capability, which should ideally be isolated from other compartments.

We test whether a new range is a subset of an existing capability's range; if so, we ignore it and return. Otherwise we continue down the list. We also test for the converse, superset case: if found, we replace the older, narrower capability and tell the caller to start scanning from the wider one just found, to cover the new regions.

This recursive process takes us from one capability (e.g., the stack pointer or a heap-resident capability) to a set expressing its transitive closure. The power of this technique is that it applies to any readable capability: those passed as arguments, left in registers, returned by functions, or residing in any memory accessible to the malicious program.

The goal is to escape a compartmentalised address space and reach the address spaces of other libraries and the main binary. If compartmentalisation is fully effective and isolates libraries from each other and the binary, this should be impossible.

```
void scan_recursive(void* cap) {
  for(int i = 0; i < get_length(cap); i++){
    access((char*)(cap+i), 0); // abuse access() to probe memory...
    if(errno == EFAULT){       // not mapped => avoid this address
        continue;
    }
    if(isValid(new_cap) && getPerms(cap) == LOAD_CAP) {
      if(!isInList(new_cap, seenHead)){
        scan_recursive(new_cap);
      }
    }
  }
}
```

Fig. 1. Code for recursive scanning.

6 Morello Linux

The first operating system analyzed was Linux running on the Morello board. We found it to be a less developed system than CheriBSD, with no compartmentalisation features beyond core CHERI capabilities. However, the example code by Brodsky et al. [3] leverages these core CHERI primitives extensively, preventing some attacks that CheriBSD addresses via specialised mitigations.

The downside is greater reliance on the developer writing secure code, rather than built-in mitigations enabled by default.

To evaluate our exploits, we use two binaries. The first is a simple binary that creates an integer on the stack or heap and prints it to the console, once before calling the malicious library and once after. The second is an example program included with Morello Linux, with an encryption key stored in a global struct; its main defence is that the struct is sealed, preventing access without the correct unsealing capability. These two programs serve as a minimal testbed for existing compartmentalisation models and what the basic Linux system provides.

The example binary is linke statically for all but one attack, while our example program is linked dynamically.

6.1 Vulnerabilities and exploit techniques

We found a number of vulnerabilities on this platform arising from different ways in which capabilities are (inadvertently) shared. There are essentially two root causes for these: the first is that when a memory region is freed, existing capabilities associated with this region may remain valid, and the second is that some APIs may leak capabilities through their normal usage. The former can be further differentiated based on whether capabilities flow from attacker to victim or vice-versa.

These vulnerabilities give rise to four exploit techniques: stack walking, heap scavenging, heap storing, and a `dlopen` infoleak. The first two exploit the capabilities left on the stack and the heap by the victim (so the capabilities flow from the victim to the attacker), whereas the third one exploits the fact that a heap chunk controlled by the attacker can potentially be re-allocated to the victim without being cleared (so the flow of capabilities is the reverse of the first two). The last exploit relies on the fact that some APIs (e.g., `dlopen`) leak sensitive capabilities. Each exploit uses the recursive scanning technique and aims to find capabilities giving the malicious library access outside its compartment. Morello Linux has fewer mitigations and less work on library-based compartmentalisation, which is one reason these attacks are effective.

Stack walking The first attack is stack walking, which makes use of the shared nature of the stack and stack pointer. A malicious library can abuse this to read the entire stack, including previous frames. Since the stack contains capabilities, the library can also read and use them, which is especially useful when they come from previous frames or compartments. Given the variety of capabilities on the stack, we rely heavily on our recursive stack walking technique. Full details of our experiments are available in the extended version [10].

Calling the recursive scanner with the stack pointer yields several capabilities with both read-write and read-execute permissions, together with the base address of `libc`. In a representative run, four capabilities are retrieved: the first with read/write permissions on a narrow range, and—more interestingly—a second with read/execute permissions spanning `libc` (verifiable against the `libc`

base via GDB or the main binary). This lets us call any `libc` function, even ones not declared in the ELF linkage. This is less useful for a malicious library, as it may create its own ELF linkages, but it is ideal for traditional code-execution attacks, letting them call any `libc` function and gain more control. The attack also breaks compartmentalisation, letting a malicious library access previous stack frames and data that should be isolated. This security issue is mitigated in CheriBSD by `c18n`'s stack hardening features.

In the example code by Brodsky et al. [3][5], data inside a global struct is sealed with a sealing capability; while the malicious code has access to the struct, it cannot use the data because it is sealed, nor access the sealing capability since that is privileged. However, stack scanning finds the sealing capability, left in an old stack frame, and we use it to unseal the data. The trusted code seals the private data capability and calls the malicious code, which iterates through the stack first for *read*/*write* capabilities, then scans them for one with *seal*/*unseal* permissions. This sealing capability is then printed and used to unseal the struct, revealing the secret. Although the secret was sealed effectively, using the sealing capability in a function call left it in a previous stack frame, where our malicious code could read it.

`Dlopen` **Infoleak** While stack walking relied on a capability being available on the stack, this attack relies on the `dlopen` function; the malicious code needs access to either the global offset table, a pointer to `dlopen`, or the dynamic linker. As shown by Hammond et al. [11], access to the dynamic linker and GOT can be withheld from the malicious code. We argue that exploits using `dlopen` are significant since the dynamic linker is part of the TCB as per Gutstein [9] and many programs make use of this interface. As before, full details of our experiments are available in the extended version [10].

```
void dlopen_infoleak(void* cap){
    Obj_Entry* handle = dlopen("libc");
    void* mapping = handle->mapbase;
}
```

Fig. 2. Pseudocode for the dlopen attack.

Although the pseudocode in Figure 2 merely calls `dlopen` and saves the return value, it obtains a capability with read/write permissions to every loaded library, including the calling binary itself. The `dlopen` documentation states that it "returns a descriptor that can be used for later references to the object" [7]. In reality, this descriptor is a pointer to an internal structure that contains a capability pointing to the base of the loaded library's `.text` section. (Note that this is unlikely to yield full write access to all code, since write access to any

[5] Under the file `src/compartments/privdata.c`

```
function heap_scavenge(void* oldArray):
  for(int i = 0; i < 1000000; i++){
    int** newChunk = malloc(0x10);
    if(newChunk == oldArray){
        printf("found %#p\n", newChunk);
        printf("read value: %#p\n", *newChunk);
        printf("%d\n", **newChunk);
        **newChunk = 0x41;
    }
    free(newChunk);
  }
}
```

Fig. 3. Code for the heap scavenge attack.

given memory also requires page-granularity permissions, which are separate from CHERI's capability permissions. Our attack is concerned with bypassing the capability layer only.)

Obtaining this capability requires only casting the pointer to the correct struct type to reveal the fields. By calling `dlopen` and reading these internal structures we find an executable capability to the main binary or the C library, bypassing compartmentalisation. This shows that stack and heap are not the only paths to bypass compartmentalisation: TCB functions like `dlopen` should not be assumed safe.

This attack is also effective against the example compartmentalisation program: we can retrieve a capability pointing to the binary's stack from the internal struct, then find the sealing capability and retrieve the secret as in the stack walking attack.

Heap Scavenging The final attacks rely on heap memory and therefore require access to an allocator. We use the default allocator in Morello Linux (musl libc's `mallocng`).

Heap scavenging attempts to find capabilities in uncleared heap memory: by repeatedly allocating, we search the heap for valid capabilities. This attack was discussed by Bramley et al. [2] in more detail, comparing several CheriBSD allocators. We show that it also works on Morello Linux and can break compartmentalisation.

The code in Figure 3 shows how this is done on Morello Linux. It simply allocates several thousand times and checks if the chunk was allocated before. Although the example is passed a known freed address with sensitive data, we can repurpose it to scan each allocated chunk for valid capabilities, which can then be fed to our recursive scanner.

Heap Storing In contrast to scavenging, the gist of this attack is to create a situation where an attacker-controlled memory chunk is freed and later re-

```
void*** heapList;
int size = 220000;
int HeapStore(){
  heapList = malloc(sizeof(void*) * size);
  for(int i = 0; i < size; i++){
    void*** new = malloc(0x10);
    heapList[i] = new;
    free(new);
  }
}
void heapCheck(void* alloc){
  for(int i = 0; i < size; i++){
    if(alloc == heapList[i]){
        printf("found %#p\n", heapList[i]);
    }
  }
}
```

Fig. 4. Code for the heap store attack.

allocated to the victim without being cleared. We show here a heap-based attack, as heap allocators tend to re-use memory regions by design (e.g., to recycle freed memory regions), leading to a more predictable usage patterns. For this exploit, the attacker creates a capability to a memory chunk through `malloc`, then frees it while retaining the capability. When the same chunk is re-allocated to the victim, the attacker will have access to the allocated chunk.

Figure 4 shows the heap storing attack. Like heap scavenging, we allocate several thousand times, but instead of checking for valid capabilities in the allocated chunks, we save the pointers and free the chunks, triggering a use-after-free. Later, when the binary calls the malicious library again (or after some time), we check whether any saved pointers now hold data. If so, the heap-storing attack succeeded and we can access live heap memory the binary is currently using.

Both heap scavening and heap storing attacks are highly dependent on application logic. For example, these attacks were unsuccessful against the example compartmentalisation code, as neither the secret key nor the sealing capability is stored in the heap. They also require the malicious code to have access to an allocator, which may not always be possible, as shown by Hammond et al. [11]. The attack also requires the program to store capabilities in heap memory without clearing or overwriting it before freeing; this is not always the case, and even heap-resident capabilities may only point to small subsets of memory, precluding a full compartment escape. Because of these limitations, heap scavenging and heap storing appear less fruitful than stack walking and the `dlopen` infoleak.

6.2 Mitigations

To mitigate stack walking, several approaches are possible. Clearing sensitive capabilities from the stack works, but requires care that zeroing is not optimised away by the compiler and is always applied. An independent stack per compartment is another effective mitigation, used by CheriBSD's c18n [8].

For the `dlopen` infoleak, we propose sealing the handle before returning. This uses existing architectural primitives and should not affect existing code, given that the handle points to an internal struct that should not be accessed directly. The remaining issue is securely storing the sealing capabilities and ensuring the handle is unsealed when used in functions such as `dlsym`. If the sealing capabilities are stored improperly, a malicious library could unseal the handle and bypass the mitigation, as shown in the stack walking attack against Morello Linux.

For heap scavenging and heap storing, the allocator could use a separate heap arena per compartment, but this might increase memory overhead, especially with many compartments. Alternative mitigations might be to zero memory before returning it from the allocator (preventing scavenging) and to use CheriBSD's heap revocation [6] (preventing heap storing).

However, if only one piece of data needs protection from a malicious library, compartmentalisation may not be required. The Morello Linux SDK example, discussed earlier (§2.1), shows how sealing can protect data: by using specific instructions to unseal data only before passing it to a function, it stays sealed by default. The sealing capability itself must obviously be protected.

7 CheriBSD

CheriBSD, a port of FreeBSD to CHERI, is the second operating system analysed. It includes several mitigations absent in Morello Linux and receives updates and security fixes more often. We focus on `c18n` [27], which confines compartments to their own stack space, and heap revocation, which prevents use-after-free [21]. These features, especially `c18n`, provide much more isolated compartments that malicious libraries must break out of.

7.1 Vulnerabilities and exploit techniques

The same four vulnerabilities exist to some extent on CheriBSD. Naturally, the OS differences and the two security features (`c18n` and heap revocation) affect the vulnerabilities. We discuss what needs to change to make each attack work on CheriBSD, what it grants the attacker, and how effective it is at breaking compartmentalisation.

Stack walking Running the same code with minor edits on CheriBSD (full output in [10]), we find several new capabilities, though the executable ones are sealed, preventing arbitrary code execution. We still obtain a capability pointing

```
typedef struct Struct_Obj_Entry {
  Elf_Size magic;      /* Magic number (sanity check) */
  Elf_Size version;    /* Version number of struct format */
  TAILQ_ENTRY(Struct_Obj_Entry) next;
  char *path;          /* Pathname of underlying file (\%) */
  char *origin_path;   /* Directory path of origin file */
  int holdcount;       /* Count of transient references */
  /* Number of times loaded by dlopen */
  int dl_refcount;
  /* Base address of mapped region */
  /* This includes the .text section and more */
  caddr_t mapbase;
  size_t mapsize;      /* Size of mapped region in bytes */
  Elf_Addr vaddrbase; /* Base address  */
  // more entries follow
}
```

Fig. 5. Internal Obj_Entry Struct in CheriBSD.

to the start of `libc`; however, unlike Morello Linux, it has read/write permissions instead of read/execute. This means we cannot call arbitrary functions and may get less use from it than on Linux, but we can potentially still modify internal `libc` variables.

However, enabling `c18n` fully prevents this attack, providing each library with a fully isolated stack. Even with recursive stack walking, our code could not find any capabilities outside the malicious compartment: all capabilities found were either sealed or pointed to this isolated stack (see [10] for the full capability list). This highlights the effectiveness of the sealing primitive for compartmentalisation.

While `c18n` is effective against stack walking, it is not a perfect compartmentalisation primitive as we shall see next.

`Dlopen` **Infoleak** As in Morello Linux, `dlopen` returns a `void*` that is actually a pointer to an `Obj_Entry` struct. Figure 5 shows the `Obj_Entry` struct: it stores a capability to the entire `.text` section in `mapbase`, and `dlopen` internally returns a pointer to this struct cast to `void*`, which is why we can access all of the struct's fields. The struct differs from Morello Linux's, but with some pointer arithmetic we obtain the `mapbase` in a similar manner, giving us read/write to the `.text` section. We only need to call `dlopen` once, as the struct is a doubly linked list, yielding the mapping of every dynamically loaded library, including the binary itself. We can then use recursive scanning to find all capabilities used by the binary, expanding the reachable address space.

Figure 6 shows how the `Obj_Entry` struct can be manipulated to obtain a capability to every loaded library, giving access to `mapbase`—a capability pointing to the base of the `.text` section and covering the entire library.

```
function exploit_dlopen():
  void* lib = dlopen("libc.so", RTLD_NOW);
  Obj_Entry* obj = (Obj_Entry*)lib;

  Obj_Entry* obj_next = TAILQ_NEXT(obj, next);
  do{
    printf("Map base: %#p\n", obj_next->mapbase);
    obj_next = TAILQ_NEXT(obj_next, next);
  } while(obj_next != NULL);
}
```

Fig. 6. Code for the dlopen attack

Heap Scavenging Heap scavenging is just as effective in CheriBSD as in Morello Linux, letting a malicious library obtain capabilities to memory outside its compartment. This is the same attack as Bramley et al. [2], who found that it is ineffective against the `snmalloc` allocator [16]. However, we find that on a typical CheriBSD configuration it can bypass compartmentalisation between a library and a binary.

Heap Storing Unlike Morello Linux, heap storing is ineffective due to the malloc revocation shim [21]. This places each freed chunk into a quarantine and, after a certain epoch, invalidates any capabilities pointing to freed heap memory. Since our attack stores pointers to freed memory, they are invalidated after the epoch and before the memory is returned to the binary.

7.2 Mitigations

The mitigations proposed for CheriBSD mirror those for Morello Linux. Existing CheriBSD's mitigations, `c18n`, and heap revocation, already effectively mitigate stack walking and heap storing and should be used.

For stack walking, `c18n` is effective, preventing our attack. It is still experimental and needs further security and performance testing beyond this paper [8]. The other three recommendations remain theoretical, as we found `c18n` by far the best mitigation.

To mitigate the `dlopen` infoleak, we propose the same protection as for Morello Linux: sealing the handle with a sealing capability known only to the dynamic linker. This would prevent a malicious program from abusing the struct to escape its compartment, but would require more bookkeeping, as the pointer must be unsealed whenever received by functions such as `dlsym`. Sealing would make the handle truly opaque to user code, preventing low-level pointer arithmetic to find capabilities.

To mitigate heap scavenging, we propose two options. First, `free` could zero memory before it can be re-allocated—this affects performance, making every

```
int main(){
  printf("in main, creating private key..");
  const int kBits = 1024;
  const int kExp = 3;
  RSA *rsa = RSA_generate_key(kBits, kExp);
  printf("rsa: %s (%#p)\n", rsa, rsa);
  malicious_library();    // call malicious library
}
...
void malicious_library():
  void* object = dlopen("libc.so.7", RTLD_NOW);
  Obj_Entry* obj = (Obj_Entry*)object;
  obj += 6;
  void* map = *obj;

  printf("testing strings...\n");
  char* testString = "BEGIN RSA PRIVATE KEY";
  for(int i = 0; i < 0x200000; i++){
    char* testKey = map+i;
    if(strlen(testKey) < 100)
        continue;
    if(strncmp(testKey,
        testString,
        strlen(testString)) == 0)
        printf("%s\n", testKey);
  }
  return; }
```

Fig. 7. Code for the binary in the proof of concept attack

`malloc` equivalent to a `calloc`. Second, using two allocators—a safe one for trusted compartments and an unsafe one for untrusted code—would help, despite impacting memory footprint. Since we have demonstrated scenarios where heap-stored capabilities are retrievable by unintended compartments, we believe this mitigation is worthwhile.

Overall, CheriBSD's mitigations are very effective, mitigating two of our attacks—a good sign for the platform's maturity.

8 Proof of Concept

Lastly, we demonstrate what breaking compartmentalisation can achieve in a proof-of-concept program, where we gain control of a private key generated by OpenSSL and stored in the binary but not given to the malicious library. The code is shown in Figure 7 and is based in part on a public contribution to StackOverflow [24].

The code creates a private key via OpenSSL, then calls our malicious library. Ideally, compartmentalisation should prevent the malicious library from accessing this key or any other data in the binary [13], but using the attacks above we can read it. The malicious library is dynamically compiled together with the binary in an isolated compartment.

The malicious program first uses `dlopen` to open any library—here `libc`, since it is always available and so `dlopen` always succeeds. We then use the extra data returned with the handle to iterate over the linked list of loaded libraries, as in the `dlopen` infoleak. Using a debugger such as GDB, we find the rough location of the private key in memory: it lies in one of the binary's heap-backed sections, outside our malicious library's compartment (full GDB output in [10]). Recursive scanning then finds a capability whose bounds include this address. Although we know the rough location, the layout of the address space is randomised at each run (by OS-level ASLR), so we still need to locate it within the section. This is easy, as the private key has a header we can scan for. Finally, we print any located keys to verify they match those generated by the binary. The full scan output [10] shows that `libssl` did not have a capability to the target section, so we had to look through other libraries; a capability to the desired section was found in `libthr` (POSIX threading library), along with 74 other capabilities. This may indicate an issue in this library: it contains far more capabilities than any other library we observed.

This attack is substantially easier because the private key has a detectable header. A raw binary key would be harder to find, but still possible: the malicious library knows the approximate memory footprint and could search for capabilities of that length. Another limitation is that the attack requires knowing the rough location of the key, as memory layouts vary between programs.

Even with compartmentalisation active and initially isolating the malicious library, we can use `dlopen` to get a capability to an internal struct whose capabilities point outside the isolated compartment. This is a weakness of library-based compartmentalisation compared to traditional IPC-based compartmentalisation: a single capability leak can let a malicious library access unintended parts of the shared address space. Put differently, the transitive closure of reachable capabilities easily becomes bigger than the programmer might expect.

8.1 Mitigations

As shown in the Morello Linux examples, sealing is among the best ways to protect data, even against attacks without current mitigations, but sealing requires secure storage. If implemented correctly, accessing the private key only through a generally sealed capability would be an effective mitigation against this proof-of-concept attack, provided that the sealing capability is stored securely. If, by contrast, it is placed in widely accessible memory, this merely adds an extra step—finding this sealing capability. Note also that the `dlopen` attack also found a second copy of the private key, so mitigating the `dlopen` infoleak is necessary to prevent our proof-of-concept attack.

9 Discussion and Conclusion

We have shown four attacks that let a malicious library escape an isolated compartment and access memory it should not be able to reach. All four work on Morello Linux, and two also work on CheriBSD with both `c18n` and heap revocation enabled. The attacks are relatively simple, and grant a means of escaping its compartment. We therefore outlined several mitigations. Some compartmentalisation claims fall short due to bugs and information leaks in the dynamic linker, the stack, and heap memory; however, this is not an architectural issue but one in the operating systems, compilers, and dynamic linkers used in CheriBSD and Morello Linux. We also recognise that compartmentalisation is still under active development, and expect our attacks to be addressed by ongoing or future developments. These vulnerabilities rely heavily on information leaks, which are much more significant on CHERI, since any pointer reveals its validity and other metadata that can be used by the attacker to recursively scan a memory range for obtaining further capabilities.

While the literature on CHERI security mitigations for common memory corruption bugs is extensive, there is less research on the compartmentalisation CHERI provides, especially against potentially malicious libraries. This paper examines the effectiveness of compartmentalisation against a malicious library and how such a library would try to break it. We find that while CHERI gives developers capabilities that protect against classic memory corruption attacks, these capabilities can still be used in insecure ways, letting a malicious library break out of an isolated compartment.

References

1. Amar, S., Chen, T., Chisnall, D., Filardo, N.W., Laurie, B., Lefeuvre, H., Liu, K., Moore, S.W., Norton-Wright, R., Seltzer, M., Tao, Y., Watson, R.N.M., Xia, H.: CHERIoT RTOS: An OS for fine-grained memory-safe compartments on low-cost embedded devices. In: Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP '25). p. 67–84. SOSP '25, Association for Computing Machinery, New York, NY, USA (2025). https://doi.org/10.1145/3731569.3764844, `https://doi.org/10.1145/3731569.3764844`
2. Bramley, J., Jacob, D., Lascu, A., Singer, J., Tratt, L.: Picking a CHERI allocator: Security and performance considerations. In: Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management. p. 111–123. ISMM 2023, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3591195.3595278
3. Brodsky, K., Khrustalev, Y., Perry, I.: Morello examples (May 2023), `https://git.morello-project.org/morello/morello-examples`
4. Eauvidoum, I., disk noise: Twenty years of escaping the Java sandbox. phrack (2021), `https://phrack.org/issues/70/7`
5. ENISA: Threat landscape for supply chain attacks. Tech. rep., ENISA (Jul 2021). https://doi.org/DOI: 10.2824/168593

6. Filardo, N.W., Gutstein, B.F., Woodruff, J., Clarke, J., Rugg, P., Davis, B., Johnston, M., Norton, R.M., Chisnall, D., Moore, S.W., Neumann, P.G., Watson, R.N.M.: Cornucopia reloaded: Load barriers for CHERI heap temporal safety. In: Gupta, R., Abu-Ghazaleh, N.B., Musuvathi, M., Tsafrir, D. (eds.) Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024. pp. 251–268. ACM (2024). https://doi.org/10.1145/3620665.3640416, https://doi.org/10.1145/3620665.3640416
7. FreeBSD: dlopen(3) manual page (2025), https://man.freebsd.org/cgi/man.cgi?query=dlopen&manpath=FreeBSD+14.3-RELEASE+and+Ports
8. Gao, D.: compartmentalization, c18n — library-based software compartmentalization (March 2024), https://man.cheribsd.org/cgi-bin/man.cgi/dev/c18n
9. Gutstein, B.: Memory safety with CHERI capabilities: security analysis, language interpreters, and heap temporal safety. Tech. Rep. UCAM-CL-TR-975, University of Cambridge, Computer Laboratory (Nov 2022). https://doi.org/10.48456/tr-975, https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-975.pdf
10. Guzairov, D., Potanin, A., Kell, S., Tiu, A.: A security analysis of CheriBSD and Morello Linux (extended version) (2026), https://arxiv.org/abs/2601.19074
11. Hammond, A., Almeida, R., Bauereiss, T., Campbell, B., Stark, I., Sewell, P.: Morello-Cerise: A proof of strong encapsulation for the Arm Morello capability hardware architecture. In: Proceedings of the ACM on Programming Languages, Volume 9 (2025). https://doi.org/https://doi.org/10.1145/3729329
12. Khan, A., Xu, D., Tian, D.J.: Low-cost privilege separation with compile time compartmentalization for embedded systems. In: 2023 IEEE Symposium on Security and Privacy (SP). pp. 3008–3025 (2023). https://doi.org/10.1109/SP46215.2023.10179388
13. Kressel, J.A., Lefeuvre, H., Olivier, P.: Software compartmentalization trade-offs with hardware capabilities. In: Proceedings of the 12th Workshop on Programming Languages and Operating Systems. p. 49–57. PLOS '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3623759.3624550
14. Ladisa, P., Plate, H., Martinez, M., Barais, O.: SoK: Taxonomy of attacks on open-source software supply chains. In: Proceedings of the 2023 IEEE Symposium on Security and Privacy (IEEE S&P 2023). pp. 1509–1526 (2023). https://doi.org/10.1109/SP50308.2023.00117, https://arxiv.org/abs/2204.04008
15. Larose, O.: Dynamic library compartmentalization. In: Companion Proceedings of the 2023 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity. p. 51–52. SPLASH 2023, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3618305.3623604, https://doi.org/10.1145/3618305.3623604
16. Liétar, P., Butler, T., Clebsch, S., Drossopoulou, S., Franco, J., Parkinson, M.J., Shamis, A., Wintersteiger, C.M., Chisnall, D.: snmalloc: a message passing allocator. In: Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management. pp. 122–135. ISMM 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3315573.3329980, https://doi.org/10.1145/3315573.3329980
17. Nagarakatte, S., Zhao, J., Martin, M., Zdancewic, S.: SoftBound: Highly compatible and complete spatial memory safety for C. In: Proceedings of the 2009

ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09). pp. 245–258 (2009). https://doi.org/10.1145/1542476.1542504
18. Reis, C., Gribble, S.D.: Isolating web programs in modern browser architectures. In: Proceedings of the 4th ACM European Conference on Computer Systems. p. 219–232. EuroSys '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1519065.1519090
19. skape: Safely searching process virtual address space (2004), https://www.hick.org/code/skape/papers/egghunt-shellcode.pdf
20. Skorstengaard, L., Devriese, D., Birkedal, L.: Reasoning about a machine with local capabilities: Provably safe stack and return pointer management. ACM Trans. Program. Lang. Syst. **42**(1) (mar 2019). https://doi.org/10.1145/3363519, https://doi.org/10.1145/3363519
21. SRI International, University of Cambridge Computer Laboratory: MRS – malloc revocation shim (2023), https://man.cheribsd.org/cgi-bin/man.cgi/dev/mrs
22. Tsoutsos, N.G., Maniatakos, M.: Anatomy of memory corruption attacks and mitigations in embedded systems. IEEE Embedded Systems Letters **10**(3), 95–98 (2018). https://doi.org/10.1109/LES.2018.2829777
23. UK Government: CHERI technology for cyber security (2025), https://www.gov.uk/government/publications/cheri-technology-for-cyber-security
24. Usman: How to generate RSA private key using OpenSSL? (2011), https://stackoverflow.com/questions/5927164/how-to-generate-rsa-private-key-using-openssl
25. Watson, R.N.M., Neumann, P.G., Woodruff, J., Roe, M., Almatary, H., Anderson, J., Baldwin, J., Barnes, G., Chisnall, D., Clarke, J., Davis, B., Eisen, L., Filardo, N.W., Fuchs, F.A., Grisenthwaite, R., Joannou, A., Laurie, B., Markettos, A.T., Moore, S.W., Murdoch, S.J., Nienhuis, K., Norton, R., Richardson, A., Rugg, P., Sewell, P., Son, S., Xia, H.: Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9). Tech. Rep. UCAM-CL-TR-987, University of Cambridge, Computer Laboratory (Sep 2023). https://doi.org/10.48456/tr-987, https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-987.pdf
26. Watson, R.N.M., Neumann, P.G., Woodruff, J., Roe, M., Anderson, J., Baldwin, J., Chisnall, D., Davis, B., Joannou, A., Laurie, B., Moore, S.W., Murdoch, S.J., Norton, R., Son, S., Xia, H.: Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 6). Tech. Rep. UCAM-CL-TR-907, University of Cambridge, Computer Laboratory (Apr 2017). https://doi.org/10.48456/tr-907, https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-907.pdf
27. Watson, R.N.M., Witaszczyk, K., Man, J.: CheriBSD 23.11 new features tutorial (2023), https://www.cheribsd.org/tutorial/23.11/c18n/index.html
28. Watson, R.N., Woodruff, J., Neumann, P.G., Moore, S.W., Anderson, J., Chisnall, D., Dave, N., Davis, B., Gudka, K., Laurie, B., Murdoch, S.J., Norton, R., Roe, M., Son, S., Vadera, M.: CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In: 2015 IEEE Symposium on Security and Privacy. pp. 20–37 (2015). https://doi.org/10.1109/SP.2015.9
29. Yu, J.Z., Watt, C., Badole, A., Carlson, T.E., Saxena, P.: Capstone: A Capability-based Foundation for Trustless Secure Memory Access (Extended Version) (Mar 2023). https://doi.org/10.48550/arXiv.2302.13863, http://arxiv.org/abs/2302.13863, arXiv:2302.13863 [cs]