

Debugging with CHERI

CHERI is designed to improve the security of both existing and future software.

John Baldwin

[CHERI](#) is a new security-oriented processor technology. Applying principles from historical capability systems, CHERI aims to improve the security of existing software written in C and C++ by extending both implicit and explicit C/C++ pointers to include metadata such as bounds and permissions. For example, the pointer returned from a call to `malloc()` on a CHERI architecture is bounded to the requested size, and attempts to access data outside of the allocation bounds will trigger a processor exception. Similarly, a pointer to an on-stack buffer is bounded to the size of the buffer when passed to another function rather than granting access to the entire thread stack. To store this extra information, CHERI defines a new hardware type: the CHERI capability.

CHERI can mitigate vulnerabilities by converting some instances of undefined behavior into crashes. While the mitigation is valuable from a security perspective, CHERI can also aid in debugging, both by providing more information to the user (such as the bounds of memory allocations) as well as catching errors sooner.

Over time, the corpus of CHERI-enabled software has grown to include a full port of FreeBSD, known as [CheriBSD](#), and several third-party applications including much of the KDE graphical desktop environment. The CHERI research team also maintains a [fork](#) of the GNU debugger with support for inspecting CHERI architectures including ARM's Morello and CHERI-RISC-V. CHERI GDB supports debugging both CheriBSD user processes and kernels. Since CheriBSD tracks FreeBSD's development, bugs reported against FreeBSD can often be reproduced and investigated on CheriBSD as well.

In this article, I investigate a couple of high-quality bug reports from FreeBSD's bug report database on CheriBSD demonstrating both the experience of debugging on CHERI as well as some of the unexpected results one can encounter with CHERI. During my investigation I used CheriBSD running on an [ARM Morello](#) system as well as CHERI images running under QEMU on my FreeBSD/amd64 desktop machine. Interactive debugging with GDB is a bit snappier on the Morello system especially for investigating userspace issues. However, QEMU can be very useful for debugging kernel bugs via its debug server. The QEMU approach can also be used on any commodity system running FreeBSD, Linux, or macOS. When working with QEMU, I used the [cheribuild](#) tool to build CheriBSD disk images as well as required tools such as LLVM, GDB, and QEMU. On the Morello system I used CheriBSD's package repository to install LLVM and GDB and used FreeBSD's standard build system to build and test new versions of base system components.

First Bug

Bug [271955](#) describes a buffer overflow in the CHAP parsing code of the [ppp\(8\)](#) utility. Beyond the very detailed writeup, this bug also includes a test program which triggers the overflow making it easy to both reproduce the bug and test potential fixes. The first steps I took were to download the source of the test program on my Morello system, compile it, and execute it. While this did fork and exec a ppp(8) process that crashed, the crash did not match the issue described in the bug. Instead, ppp(8) had crashed much

earlier, as seen in Figure 1. One of the first things to notice, in comparison to using GDB on a non-CHERI architecture, is that pointers are annotated with additional data in square brackets. For example, the `path` argument to the `ID0open()` function is annotated by the string `[rwRW,0x1d8aa0-0x1f1f60]`. The first field before the comma indicates the permissions of the pointer. In this case the pointer can be used to read and write both data (lowercase letters) and other pointers (uppercase letters). The address range after the command gives the bounds of this pointer describing the range of accessible addresses. Another detail to note is that this crash was associated with a new signal, `SIGPROT`, used by CheriBSD for CHERI exceptions. In this case, a memory operation used an invalid capability which led to the crash. The pointer dereference may not be obvious in this line of code, but the invocation of `va_arg()` is in fact an implicit memory reference. Looking at the surrounding code in Figure 2, we see that this function assumes there is always a third integer argument after the `flags` argument, which seems a bit dodgy. The point of variadic arguments is that they are not always present. Walking up the stack to examine the caller in Figure 3, the caller is in fact only passing two arguments to `ID0open()`. Trying to fetch a non-existent variadic argument is Undefined Behavior (UB) in C; it just happens to be more fatal on CHERI than on other architectures FreeBSD supports.

Figure 1: Crash Summary from GDB

```
Core was generated by `ppp'.
Program terminated with signal SIGPROT, CHERI protection violation.
Capability tag fault.
#0  0x000000000195ed0 in ID0open (
    path=0x1d8af4 <bundle_Create.bundle+84> [rwRW,0x1d8aa0-0x1f1f60] "/dev/tun0",
    flags=flags@entry=2)
    at /usr/home/john/work/git/cheribsd/usr.sbin/ppp/id.c:148
148      ret = open(path, flags, va_arg(ap, int));
```

Figure 2: Listing of ID0open Source

```
(gdb) l
143      int ret;
144      va_list ap;
145
146      va_start(ap, flags);
147      ID0set0();
148      ret = open(path, flags, va_arg(ap, int));
149      log_Printf(LogID0, "%d = open(\"%s\", %d)\n", ret, path, flags);
150      ID0setuser();
151      va_end(ap);
152      return ret;
```

Figure 3: Caller of ID0open

```
(gdb) up
#1  0x000000000158f90 in bundle_Create (
    prefix=0x1322b8 [rR,0x1322b8-0x1322c1] "/dev/tun", type=1,
    type@entry=1254072, unit=unit@entry=-1)
    at /usr/home/john/work/git/cheribsd/usr.sbin/ppp/bundle.c:726
726      bundle.dev.fd = ID0open(bundle.dev.Name, O_RDWR);
```

From the code listed in Figure 2 and the function name, it is apparent that the purpose of this function is to wrap the [open\(2\)](#) system call, executing open(2) as root. open(2) is documented as taking an optional third argument if the O_CREAT flag is passed, so the [fix](#) for this particular bug in ppp(8) is to only fetch the variadic argument when O_CREAT is specified. With that issue out of the way, it's time to get back to debugging the original bug.

After re-running the test program with a fixed ppp(8) binary, I was greeted with the kernel crash from Figure 4. To debug this more easily, I switched to testing with QEMU on my FreeBSD/amd4 desktop system. I started the GDB

server in QEMU via the `-s` command line argument and used `gdb` to connect to the QEMU guest using a native CHERI GDB binary built by the `kgdb-native` `cheribuild` target.

Figure 4: Kernel Crash Messages

```
panic: Capability abort from kernel space: bounds violation
cpuid = 0
time = 1779294044
KDB: stack backtrace:
db_trace_self() at db_trace_self
db_trace_self_wrapper() at db_trace_self_wrapper+0x3c
vpanic() at vpanic+0x1a0
panic() at panic+0x48
cap_abort() at cap_abort+0x288
handle_ellh_sync() at handle_ellh_sync+0x18
--- exception, esr 0x9600002a
_memcpy() at _memcpy+0xf4
make_packet() at make_packet+0x90
pppoe_start() at pppoe_start+0x180
ng_pppoe_rcvmsg() at ng_pppoe_rcvmsg+0xba4
ng_apply_item() at ng_apply_item+0x348
ng_snd_item() at ng_snd_item+0x364
ngc_send() at ngc_send+0x1f4
```

Once connected to the guest I walked up the stack to the frame of the `make_packet()` function. Examining the arguments passed to `bcopy()` in Figure 5, the code is trying to copy 12 bytes from a buffer that is only 4 bytes long. Reading the rest of the code in the loop around the `bcopy()`, it seems evident that these tag structures have a 4 byte header containing a type and length (in network byte order) followed by a data buffer whose length is given in the header. The `hlen` variable contains the total length, so why were the bounds of the pointer too small?

Figure 5: make_packet() Stack Frame

```
#18 0xffff0000b0c90b04 in make_packet (
    sp=0xfffffa0000180bd80 [rwRW,0xfffffa0000180bd80-0xfffffa0000180be00])
    at /usr/home/john/work/git/cheribsd/sys/netgraph/ng_pppoe.c:431
431                bcopy(*tag, (char *)dp, tlen);
(gdb) p *tag
$1 = (const struct pppoe_tag *) 0xfffffa0000112ee74 [rwRW,0xfffffa0000112ee74-0xff
ffa0000112ee78]
(gdb) p tlen
$2 = 12
```

My next step was to find out where these tag pointers came from. The loop in `make_packet()` is walking an array of tag pointers stored in `sp->neg->tags`. Searching in the `ng_pppoe.c` source file led me to the `insert_tag()` function which adds a tag pointer to this array. The next step then was to figure out where the tag pointers for this packet came from. Moving up to the caller's stack frame and listing the surrounding code showed the origin of these tags via several calls to `insert_tag()` in `pppoe_start()` in Figure 6.

Figure 6: pppoe_start() Stack Frame

```
(gdb) up
#19 0xffff0000b0c90868 in pppoe_start (
    sp=0xfffffa0000180bd80 [rwRW,0xfffffa0000180bd80-0xfffffa0000180be00])
    at /usr/home/john/work/git/cheribsd/sys/netgraph/ng_pppoe.c:1305
1305        make_packet(sp);
(gdb) l
1300        init_tags(sp);
1301        insert_tag(sp, &neg->host_uniq.hdr);
1302        insert_tag(sp, &neg->service.hdr);
1303        if (privp->max_payload.data != 0)
1304            insert_tag(sp,
```

```

                                &privp->max_payload.hdr);
1305         make_packet(sp);
1306         /*
1307          * Send packet and prepare to retransmit it after timeout.
1308          */
1309         ng_callout(&neg->handle, node, hook, PPPOE_INITIAL_TIMEOUT * hz,

```

At this point, I had an aha moment. In CHERI, bounds for pointers are narrowed to the size of a logical allocation. For example, a pointer returned from `malloc()` is able to access the entire allocated buffer, and pointers to global variables or stack variables permit access to the entire variable. If a variable contains a structure with multiple fields, a pointer to that variable is permitted to access all of the fields. Similarly, if a variable contains an array of objects, a pointer to the variable can access all of the elements in the array. Using the address-of operator (`&`) to take the address of a field or array element generates a pointer with a potentially different address (to account for the offset of the field or array element), but with the same bounds and permissions.

CHERI LLVM implements an optional feature called subobject bounds that changes the behavior of the address-of operator (`&`) to narrow the bounds of the resulting pointer to permit access only to the individual field or array element. In some cases this can be useful. For example, if a structure contains an array of characters that holds a string, narrowing the bounds to only the array of characters when passing a pointer to that structure field to `strcpy()` can protect against an overflow into other structure fields. However, there is also a fair bit of real-world C code that depends on taking a pointer to a field and later casting it back to a pointer to the containing object. An example of this is the `container_of()` macro used in Linux. A similar pattern exists in some of the macros from `<sys/queue.h>` on the BSDs. Another common pattern in C is the implementation of C++-style OOP by defining a structure containing the fields of a base class and embedding an instance of that structure as the first field in structures defining subclasses. This is often paired with the use of function pointers to implement virtual methods. The

function pointers accept pointers to the base class structure type but cast them to the subclass structure type. Several subsystems in FreeBSD's kernel make use of this pattern. Because these common idioms of C are not compatible with subobject bounds, they are not enabled in CHERI LLVM by default. However, CHERI LLVM does permit using source code annotations to disable subobject bounds for specific types or for specific fields of a type. When a field has been annotated in this way, the address-of operator does not narrow the bounds of the resulting pointer. CheriBSD enables subobject bounds for the kernel both to provide increased security and to be a research case study for evaluating subobject bounds. In some cases, such as the use of the `container_of()` macro, CHERI LLVM is able to emit a compile time warning if a field is not marked with an annotation disabling subobject bounds. However, for other cases, the compiler is not able to detect when the address-of operator is used to generate a pointer that is later used to access the containing structure. Unfortunately, those cases are currently found by encountering a bounds fault panic at runtime when exercising the relevant code path.

Which brings us back to the panic in the `ng_pppoe` module. Here, the module is using a variant of the subclass model described above where the `struct pppoe_tag` type defines the common fields at the start of each PPPoE tag. The module defines two additional types, `struct datatag` and `struct maxptag`, that each embed a `struct pppoe_tag` object as the first field (named "hdr") followed by additional fields holding tag-specific data. When `pppoe_start()` calls `insert_tag()` to insert each of the tags it uses into the array of tag pointers shown in Figure 6, it is using the address-of operator to take a pointer to the `hdr` field of a `struct datatag`. As a result, the bounds of the pointer are reduced to the 4 byte header structure. Note that it is ok to narrow the bounds to an individual `struct datatag`, just not the header. For example, for the expression `&neg->host_uniq.hdr`, it is ok to narrow the bounds to the `host_uniq` member of the `neg` structure, it is only the further narrowing to the `hdr` field that is problematic. To fix this, I just needed to add a few source code annotations to disable subobject bounds for the `hdr` member. With that fix applied, it was time to run the test program once again.

Finally, I am now able to reproduce the original crash reported in the bug, and indeed the crash shown in Figure 7 is very similar. One difference from the crash reported in the bug is that the ChERI version died with a ChERI fault due to reading from an untagged pointer. However, if the original bug is a buffer overflow as the bug report suggests, why didn't ChERI detect the buffer overflow and crash sooner? According to the bug report, the overflow happens when the `chap_Input()` function reads the CHAP challenge into the `chap->challenge.peer` array. Indeed, when we walk up the stack and examine the data structure containing the invalid pointer in Figure 8, it is obvious that the `peer` array and several following fields up to the end of the structure are filled with bytes containing `0xff`. So why didn't ChERI catch the overflow of the `peer` array? The answer, it turns out, is once again subobject bounds.

Recall that subobject bounds are not enabled by default in ChERI LLVM, and that CheriBSD only enables them for the kernel, not for userspace. So, when the `chap_Input()` function obtains a pointer to the `peer` array, that pointer has the bounds of the containing object, in this case the entire `authp->physical->dl` object. Had the overflow written off the end of the `dl` object, ChERI would have raised a bounds violation exception, but the test program was careful to overflow the `peer` array while staying within the bounds of the containing object.

Figure 7: Second Crash Summary from GDB

```
Core was generated by `ppp -nat -foreground pppoe-client'.
Program terminated with signal SIGPROT, ChERI protection violation.
Capability tag fault.
#0  strlen (
    str=0x11f [rwxRWE,0xffffffffffffffff-0xffffffffffffffff]
    (invalid,sealed) <error: Cannot access memory at address 0x11f>)
    at /usr/home/john/work/git/cheribsd/lib/libc/string/strlen.c:143
143             while (*p != '\0')
```

Figure 8: Investigating the Invalid Pointer

```
(gdb) up
#1  0x000000000160ac4 in chap_ChallengeInit (
    authp=0x42a74a30 [rwRW,0x42a72000-0x42a75000])
    at /usr/home/john/work/git/cheribsd/usr.sbin/ppp/chap.c:493
493      len = strlen(authp->physical->dl->bundle->cfg.auth.name);
(gdb) p authp->physical->dl->bundle
$2 = (struct bundle *) 0xffffffffffffffff
      [rwxRWE,0xffffffffffffffff-0xffffffffffffffff] (invalid,sealed)
(gdb) p authp->physical->dl
$3 = (struct datalink *) 0x42a72000 [rwRW,0x42a72000-0x42a75000]
(gdb) p *authp->physical->dl
$4 = {desc = {type = 5,
...
    local = "\000", '\245' <repeats 147 times>,
    peer = '\377' <repeats 148 times>}, NTRespSent = 1,
    peertries = -1,
    authresponse = '\377' <repeats 48 times>}, mp = {seq = 4294967295,
    bandwidth = 4294967295},
    bundle = 0xffffffffffffffff
      [rwxRWE,0xffffffffffffffff-0xffffffffffffffff] (invalid,sealed),
    next = 0xffffffffffffffff
      [rwxRWE,0xffffffffffffffff-0xffffffffffffffff] (invalid,sealed)}
```

After confirming the bug was indeed an overflow of the `peer` array, the most obvious fix was to add a bounds check on the challenge length and reject challenges with an invalid length. I was curious if there was a standardized limit on CHAP challenges. A quick web search led me to [RFC 1994](#). Reading the description of the Challenge packet, there is no limit to the length of the Value field other than the implicit limit of 255 characters since that is the maximum value that can be specified in the single octet Value-Size field. ppp(8) was using an artificially low limit of 48 characters, so the fix I pursued

was to increase the maximum challenge length to 255 to support any valid challenge.

Second Bug

Bug [271820](#) describes another buffer overflow from the NAT handling of the ppp(8) daemon. In particular, ppp(8)'s NAT handling contains a few handlers for specific network protocols, and this bug describes a problem in the handler for the NetBIOS over TCP/IP datagram service. Despite the name, this protocol uses UDP. The specific bug is related to IP fragments whereby a large UDP message can be split into multiple IP packets. Each of these smaller IP packets, or fragments, contain a subset of the original UDP datagram. The NetBIOS handler attempts to parse the contents of the UDP payload and is careful to avoid reading off the end of the UDP packet. The handler does assume that the packet being examined contains the entire UDP payload. However, if the packet being examined is an IP fragment, it may contain only a subset of the UDP payload.

As with the previous bug, this bug report contains a test program which reproduces the buffer overflow. However, the submitter notes that this particular overflow does not reliably crash the ppp(8) process when executed on a conventional (non-CHERI) architecture. When the test program is run on a CHERI system, ppp(8) reliably crashes with a bounds fault when the overflow occurs as shown in Figure 9. This makes it easier to validate potential fixes, as ppp(8) will no longer crash once the overflow is prevented. One curious thing to note is that the `pmax` argument to `AliasHandleName()` is marked as invalid. If we examine the start of the `AliasHandleUdpNbtNs()` function, both `p` and `pmax` are ultimately computed by adding an offset to the start of the IP packet as shown in Figure 10. Thus, both pointers should have the same metadata, such as bounds and permissions. However, the bounds on `pmax` cover a different range than `p`, though both ranges have the same length. In CHERI, bounds are represented as a bounded range near the current capability address. This allows for an efficient representation of these bounds in the extra word of metadata used by CHERI capabilities, but it does

mean that if a pointer's address is moved sufficiently far from its original address, the metadata can no longer describe the correct bounds. When this happens, CHERI capabilities are marked invalid, hence why `pmax` is marked invalid even though it is computed as an offset against a valid pointer. While `p` also has an offset relative to the original start of the packet, it is still close enough to the original pointer's address that it is marked valid even though `p` itself is out of bounds.

Figure 9: Third Crash Summary from GDB

```
Core was generated by `ppp -nat -direct'.
Program terminated with signal SIGPROT, CHERI protection violation.
Capability bounds fault.

#0  AliasHandleName (p=0x42b52419 [rwRW,0x42b51000-0x42b52400] "",
    pmax=0x42b6114f [rwRW,0x42b61000-0x42b62400] (invalid) "")
    at /usr/home/john/work/git/cheribsd/sys/netinet/libalias/alias_nbt.c:268
268      while ((*p & 0x3f) != 0x00) {
(gdb) where
#0  AliasHandleName (p=0x42b52419 [rwRW,0x42b51000-0x42b52400] "",
    pmax=0x42b6114f [rwRW,0x42b61000-0x42b62400] (invalid) "")
    at /usr/home/john/work/git/cheribsd/sys/netinet/libalias/alias_nbt.c:268
#1  AliasHandleQuestion (count=32673, q=<optimized out>,
    pmax=0x42b6114f [rwRW,0x42b61000-0x42b62400] (invalid) "",
    nbtarg=0x44e8af80 [rwRW,0x44a8b000-0x44e8b000])
    at /usr/home/john/work/git/cheribsd/sys/netinet/libalias/alias_nbt.c:424
#2  AliasHandleUdpNbtNS (la=<optimized out>, pip=<optimized out>,
    lnk=<optimized out>, alias_address=<optimized out>,
    alias_port=0x43dffe50 [rwRW,0x43dffe50-0x43dffe52],
    original_address=0x43dffe58 [rwRW,0x43dffe58-0x43dffe5c],
    original_port=0x42b51152 [rwRW,0x42b51000-0x42b52400])
    at /usr/home/john/work/git/cheribsd/sys/netinet/libalias/alias_nbt.c:805
```

Figure 10: Computation of pmax

```
(gdb) l AliasHandleUdpNbtNS
...
776
777         /* Calculate data length of UDP packet */
778         uh = (struct udphdr *)ip_next(pip);
779         nbtarg.uh_sum = &(uh->uh_sum);
780         nsh = (NbtNSHeader *)udp_next(uh);
781         p = (u_char *) (nsh + 1);
782         pmax = (char *)uh + ntohs(uh->uh_ulen);
783
784         if ((char *) (nsh + 1) > pmax)
785             return (-1);
```

Reading the code of the various functions in the `alias_nbt.c` source file, they are generally well-behaved in that they honor the constraints of only reading data between `p` and `pmax` and aborting anytime an access would be attempted beyond `pmax`. The bug here, then, is that `pmax` is simply set to the wrong value in this case. The fix is to also compute the end of the underlying IP packet using the length field from the IP header and use the resulting pointer as the value of `pmax` if it is less than the original value of `pmax`. With this fix in place, the test program no longer triggers a crash in `ppp(8)`. Oddly enough, there are two functions in `alias_nbt.c` with this same `pmax` computation: `AliasHandleUdpNbt()` and `AliasHandleUdpNbtNs()`. In my first attempt at fixing this, I only patched the first function by mistake, which resulted in `ppp(8)` continuing to crash. Once I realized my error, I patched both functions in the final fix.

Lastly, the bug reporter also noted that the reproducer used an invalid IP header length of 0 that the `libalias` code did not detect. I also wrote a followup patch to add additional validation of the IP header including both the version and length fields before the packet was passed to other handlers in the library. Since this particular bug did not trigger a crash, I relied on

stepping through the `ppp(8)` process using GDB to validate that the packet was rejected earlier in the `libAliasInLocked()` function.

Conclusion

CHERI is designed to improve the security of both existing and future software. A key component of CHERI is the capability hardware type and its use as an implementation of language-level pointers in C and C++. A side effect of this use of CHERI capabilities is an additional level of detail available to debuggers at runtime. Some aspects of these details require understanding of CHERI concepts such as pointer representability to interpret.

Another takeaway is the value of high-quality bug reports. Both of the bug reports examined in this article provided a detailed, thorough analysis of the underlying bug as well as an easy way for others to reproduce the problem. These details make it much easier to reason about a solution to the problem and validate potential fixes.

John Baldwin is a systems software developer. He has directly committed changes to the FreeBSD operating system for over twenty years across various parts of the kernel (including x86 platform support, SMP, various device drivers, and the virtual memory subsystem) and userspace programs. In addition to writing code, John has served on the FreeBSD core and release engineering teams. He has also contributed to the GDB debugger. John lives in Ashland, Virginia with his wife, Kimberly, and three children: Janelle, Evan, and Bella.

Copyright © 2026 Held by Owner/Author. Publication Rights Licensed to The FreeBSD Journal. This work is licensed under a [**Creative Commons Attribution International 4.0 License**](#).

3980 Broadway Street

STE #103-107

Boulder CO 80304

1-720-207-5142

info@freebsd.foundation.org

