Porting Software to CHERI

Cybersecurity by design - from research to industry
Cheltenham, 2024-11-12

Dr Graeme Jenkinson
Director of Applied Technology | Capabilities Limited

CAPABILITIES LIMITED

Today’s talk

What does it even mean to port software to CHERI?

What kinds of changes are required and how much effort does that involve?

And when I do all this what is achieved?

What does it even mean to port software to CHERI?

“**porting** is the process of adapting **software** for the purpose of achieving some form of execution in a computing environment

… the term "port" is derived from the Latin portare, meaning "to carry". When code is not compatible with a particular operating system [*or language*] or architecture, the code must be "carried" [or "ported"] to the new system.”

- Wikipedia, Porting [Software Engineering]

Porting to Memory Safe languages

"70% of security vulnerabilities that Microsoft fixes and assigns a CVEs are due to memory safety issues. This is despite mitigations including intense code review, training, static analysis, and more."

https://msrc.microsoft.com/blog/2019/07/we-need-a-safer-systems-programming-language/

Programming language	Approximate LoC (Open Source projects)	Memory safety	Memory safety with CHERI
C	10,000,000,000	✗	✚
C++	3,000,000,000	✗	✚
Rust	40,000,000	✚	✚ ✚

Synopsys Black Duck Open Hub: https://www.openhub.net/languages?query=rust&sort=code

Porting to CHERI

For this talk we are focussed on aspects of porting related to the CHERI architecture.

- And specifically porting legacy C/C++ codebases.

Design goals:

1. C programmers should be able to port existing C code bases with minimal effort.
2. Existing compiler infrastructure and optimisations should require only limited changes.
3. Memory-safety errors that can lead to exploitable vulnerabilities should be mitigated where possible.

Conventional architecture C/C++

```
unsigned long long
incrementInteger(unsigned long long num) {
    return num + 1;
}

char*
incrementPointer(char* ptr) {
    return ptr + 1;
}
```

```
incrementInteger(unsigned long long):
        sub     sp, sp, #16
        str     x0, [sp, 8]
        ldr     x0, [sp, 8]
        add     x0, x0, 1
        add     sp, sp, 16
        ret

incrementPointer(char*):
        sub     sp, sp, #16
        str     x0, [sp, 8]
        ldr     x0, [sp, 8]
        add     x0, x0, 1
        add     sp, sp, 16
        ret
```

Conventional C: Pointers represented with simple machine-word integers

https://godbolt.org/

What is CHERI C/C++?

```
unsigned long long
incrementInteger(unsigned long long num) {
   return num + 1;
}

char*
incrementPointer(char* ptr) {
   return ptr + 1;
}
```

```
incrementInteger(unsigned long long):
      sub   csp, csp, #16
      str   x0, [csp, #8]
      ldr   x8, [csp, #8]
      add   x0, x8, #1
      add   csp, csp, #16
      ret   c30

incrementPointer(char*):
      sub   csp, csp, #16
      str   c0, [csp, #0]
      ldr   c0, [csp, #0]
      add   c0, c0, #1
      add   csp, csp, #16
      ret   c30
```

“Pointer arithmetic is implemented as arithmetic over these capabilities”

“Basic idea is to represent all C source-language pointers with machine capabilities, instead of machine words” https://github.com/chromium-cheri/chromium/commit/a39b2fe1f1e2c4a84c68aa9045637f39fccd6f16#diff-6d5f6348e30b4627ecd30be810c82c394d27e0877483789de3a497416f99a151R25

Representation of C language pointers with capabilities

```
struct DataOrder {
 DataType type;
 uint64_t value;
};
```

→

```
struct DataOrder {
 DataType type;
 uintptr_t value;
};
```

Modify type usage to ensure pointer and integer values are distinct

What can programmers rely on and what they are required to ensure, for well defined CHERI C/C++?

In the presence of compiler optimisations, this can be complex

However, in practice most things that programmers are required to do is straightforward following a set of common porting tasks:

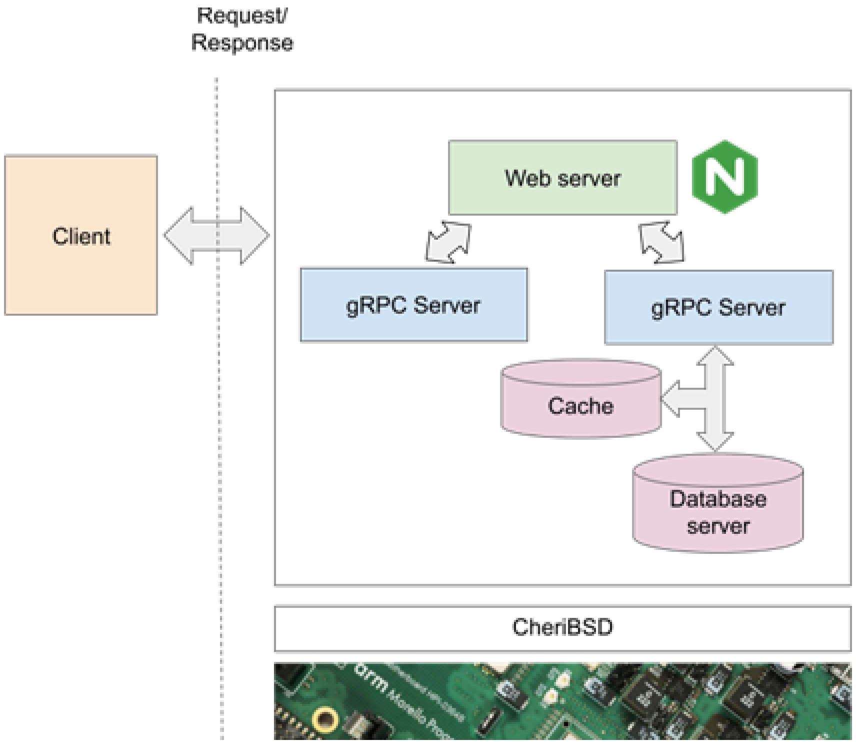
See CHERI C/C++ programming guide: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-947.pdf

Example nginx web server

A web server accepts requests via HTTP or its secure variant HTTPS. A user agent, commonly a web browser, initiates communication by making a request for a resource, and the server responds with the content or an error.

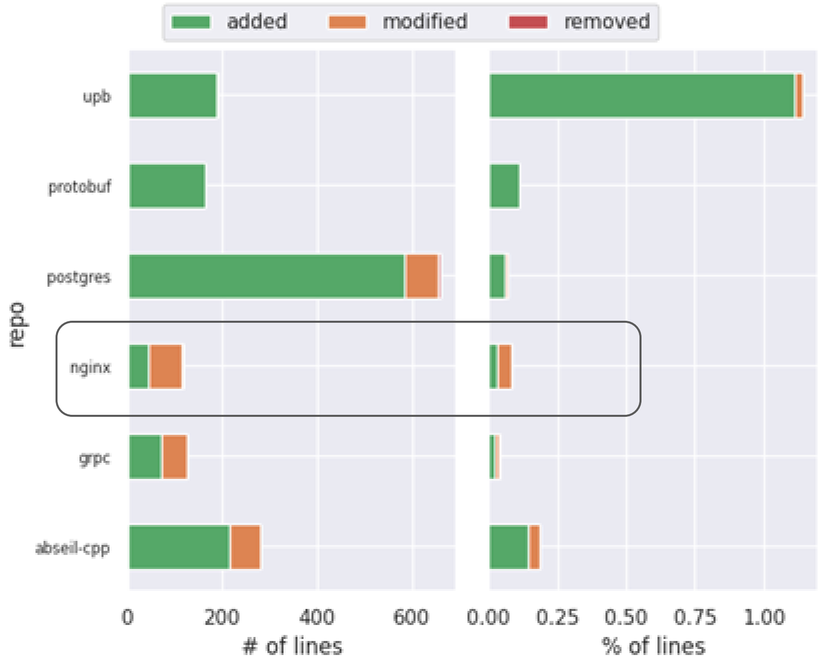
nginx is currently most deployed web server accounting for 34.1%; can also be deployed as a reverse proxy, load balancer, mail proxy and HTTP cache.

Approximately 140k lines of C code.



How much effort does that involve?

- %SLoc changed in nginx port approximately 0.10%
- Consistent with other recent studies with %SLoC changes typically 0.10%-0.25%
- Limitations:
  - The nginx port is fairly mature, but further issues may arise with testing.
  - nginx memory allocators must be modified to obtain the full benefits of CHERI spatial memory protection.
  - Modifications to support sub-object bounds are missing from these estimates.



Project	Total SLoC	Changed SLoC	% Changed SLoC	Total files	Changed files	% Changed files
nginx w/o tests	139804	118	0.10	337	20	5.90

Sliding scale of Effort

Non effort (0%)

Desktop stack - Plasma-framework, Dolphin,

- Modern C/C++ usage across code base
- Use of C++ where templating reduces us of integer/pointer conversions
- High-level applications, rather than low-level software

Minimal/small effort (0.10-0.25%)

Web service stack - nginx, Postgres, protobuf

High-effort (1-2%)

Operating system kernels - FreeBSD

Language runtimes - v8 Javascript runtime

- Modern C/C++ usage
- Misuse of standard types
- Complex memory allocators
- Internal Memory models

What is achieved?

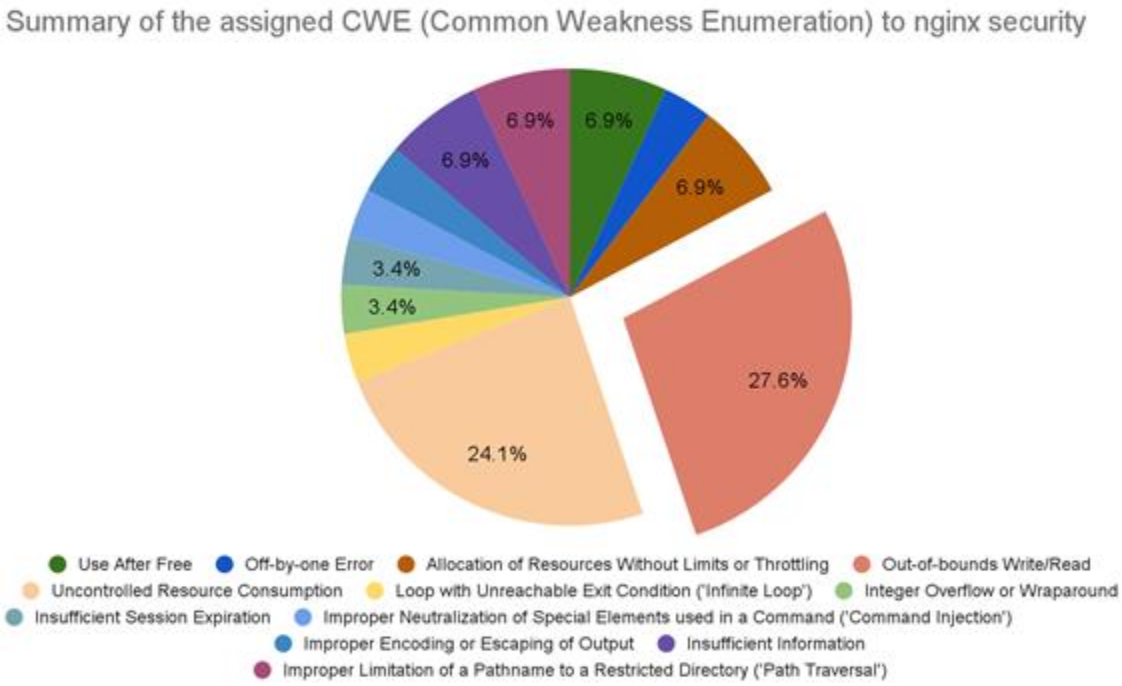
Threat model:

- Remote code execution
- Private data disclosure
- Denial of service

Analytical study analysing historic vulnerabilities

Approximately 28% of CWEs assigned to nginx security advisories relate to buffer overwrites and overreads

Uncontrolled resource consumption is the second largest weakness



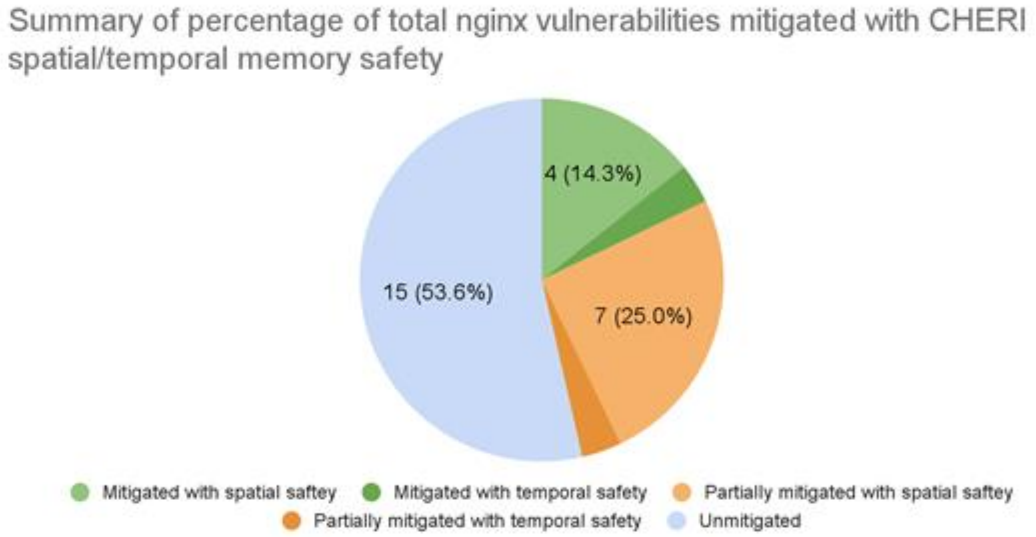
What is achieved?

CHERI protections have been shown to mitigate ~60-70% of memory safety vulnerabilities

- Memory safety issues accounting for around 70% of the total vulnerabilities

Mitigation rate of security vulnerabilities in nginx with CHERI spatial/temporal memory protection is approximately 46%

Applying compartmentalisation to nginx modules improves the potential total mitigation rate to 61%



Q&A

What does it even mean to port software to CHERI?

Porting to CHERI C/C++.

What kinds of changes are required and how much effort does that involve?

Typically in the region of 0.1-0.25%; larger for some classes of software.

And when I do all this what is achieved?

Deterministic mitigation of approximately 60-70% of memory safety issues.