C++ on the Web

Ponies for developers
Without pwn'ing users

@jfbastien

These slides: https://goo.gl/3WVQYN

Talk given at CppCon 2015 by JF Bastien.

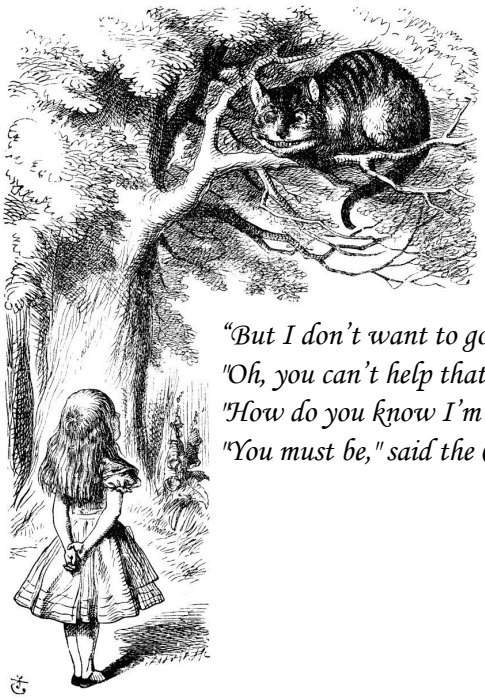
Abstract:

Is it possible to write apps in C++ that run in the browser with native code speed? Yes. Can you do this without the security problems associated with running native code downloaded from the net? Yes and yes. Come to this session to learn how.

We'll showcase some resource-intensive applications that have been compiled to run in the browser. These applications run as fast as native code with access to cornerstone native programming APIs—modern C++ STL, OpenGL, files and processes with full access to C++'s concurrency and parallelism—all in an architecture- and OS-agnostic packaging. Then, we'll describe how we deliver native code on the web securely, so developers get their C++ ponies and users don't get pwn'd. We'll also touch on the fuzzing, code randomization, and sandboxing that keep the billions of web users safe.

About JF Bastien:

JF Bastien is a compiler engineer and tech lead on Google's Chrome web browser, currently focusing on performance and security to bring portable, fast and secure code to the Web. JF is a member of the C++ standards committee, where his mechanical engineering degree serves little purpose. He's worked on startup incubators, business jets, flight simulators, CPUs, dynamic binary translation, systems, and compilers.



"But I don't want to go among people who execute C++ on the web," Alice remarked.
"Oh, you can't help that," said the Cat: "we're all mad here. I'm mad. You're mad."
"How do you know I'm mad?" said Alice.
"You must be," said the Cat, "or you wouldn't have come to a C++ conference."

— *Lewis Carroll, Alice in Wonderland*

Image credits: John Tenniel, from Alice in Wonderland.

You've heard of insecurity in NPAPI, ActiveX. Java plugins updates all the time. You know a single C++ undefined behavior can lead to demons flying out of your nose. Surely, letting arbitrary C++ run inside a browser is a bad idea!

C++? In my browser? It's more likely than you think!

Image credits: John Tenniel, from Alice in Wonderland.

Goals

- Demo C++ & web: it's a thing, you want it
- Convince you: not a bad idea for security
- Security: for your code too

Image credits: John Tenniel, from Alice in Wonderland.

This talk goes over the work of *many* people, especially in Chrome's NaCl team, Chrome security and Google technical infrastructure.
I'm but a small contributor to this!

I also discuss Chrome because I work for Google. Other browsers do similar things, sometimes better than Chrome and sometimes worse. Browser implementors do talk to each other, and borrow from one another.

Image credits: John Tenniel, from Alice in Wonderland.

I'm reusing a demo built for a former incarnation of this presentation to showcase what we want to web platform to be able to do.

The demo showcases an in-browser development environment:

- `bash`
- `ls`
- `touch foo`
- `echo "Hello, World!"`
- `cat` a file that contains an ASCII cat
- None of this is "fake": these are the actual GNU tools, compiled mostly as-is.
- Then I start executing Python:
  - Write a fibonacci function
  - Show it working on a small number
  - Show it being sad on a large number, but being interruptible with CTRL+C, this seems trivial but handling a signal is actually quite complex!
  - Show `import datetime` and `datetime.datetime.now()` and what such a simple program actually involves

Step back a bit, and explain how the development environment works:

- JavaScript is our µkernel: it spawns the processes, and sets up the "pipes"

- between processes and the console.
  - Show `./jseval -e 'alert(1+1)'`
  - Explain process creation through embed tag creation
  - Explain pipe emulation through postMessage
  - Explain how the HTML5 filesystem works, how a standard-based filesystem could work similarly, how other filesystems can be used with mount such as Google Drive or Dropbox
- Show `git clone https://github.com/jfbastien/papers.git`
  - Usage of sockets
  - Usage of SSL, and certificate verification
  - Remember: git uses a few C programs glued together with Perl scripts!

We need to go deeper:

- `cd demo1`
- `vim Makefile`
- `make -j4`
- Why yes, I use `-j4` and it actually spawns multiple processes.
- `./hello.pexe`
- `make fire` demonstrates interacting with the GPU. GLES3 is coming to browsers soon.

And now for something completely different:

- `emacs` and write a C++ fibonacci program which uses cstdint
  - Note how emacs takes forever to start... That's because it's setting itself up, parsing all of its elisp (I use the same setup here as what I use at work). Usually, emacs sets itself up once, then coredumps itself and subsequent loads reuse this "serialized" version of emacs. My demo currently doesn't support this neat hack.
- `clang fib.cc -o fib.ll -emit-llvm -S -O2`
- clang errors out because `cstdint` is a C++11 header
- `clang fib.cc -o fib.ll -emit-llvm -S -O2 -std=c++14`
- Show emacs with `llvm-mode` syntax highlighting by opening `fib.ll`
- `llc -march=js -filetype=asm fib.ll -o fib.js`
- I just transformed the bitcode file to JavaScript using a hand-rolled version of Emscripten, running inside my browser using PNaCl. Crowd goes wild (I expect readers to cheer as well, that would be you).

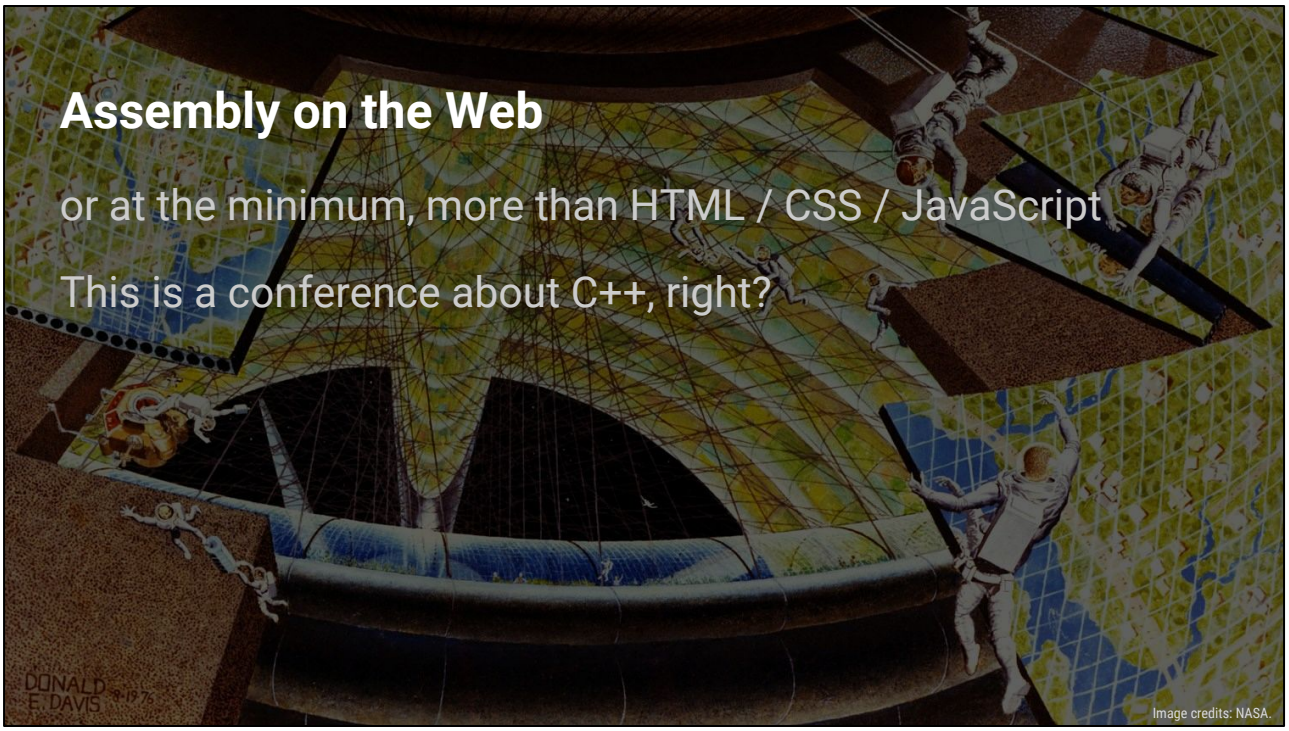
By the way, I've been serving this entire demo from the installed Chrome app using `python -m SimpleHTTPServer`, from the git repo of this presentation cloned into the app's storage, on a Chromebook Pixel 2.

X also works:

- Start the X server application
- `DISPLAY=:42 xeyes`
- `DISPLAY=:42 emacs`

I've seen a demo of the GIMP image editor working in there! It's unfortunately not currently available.

Image credits: JF Bastien.



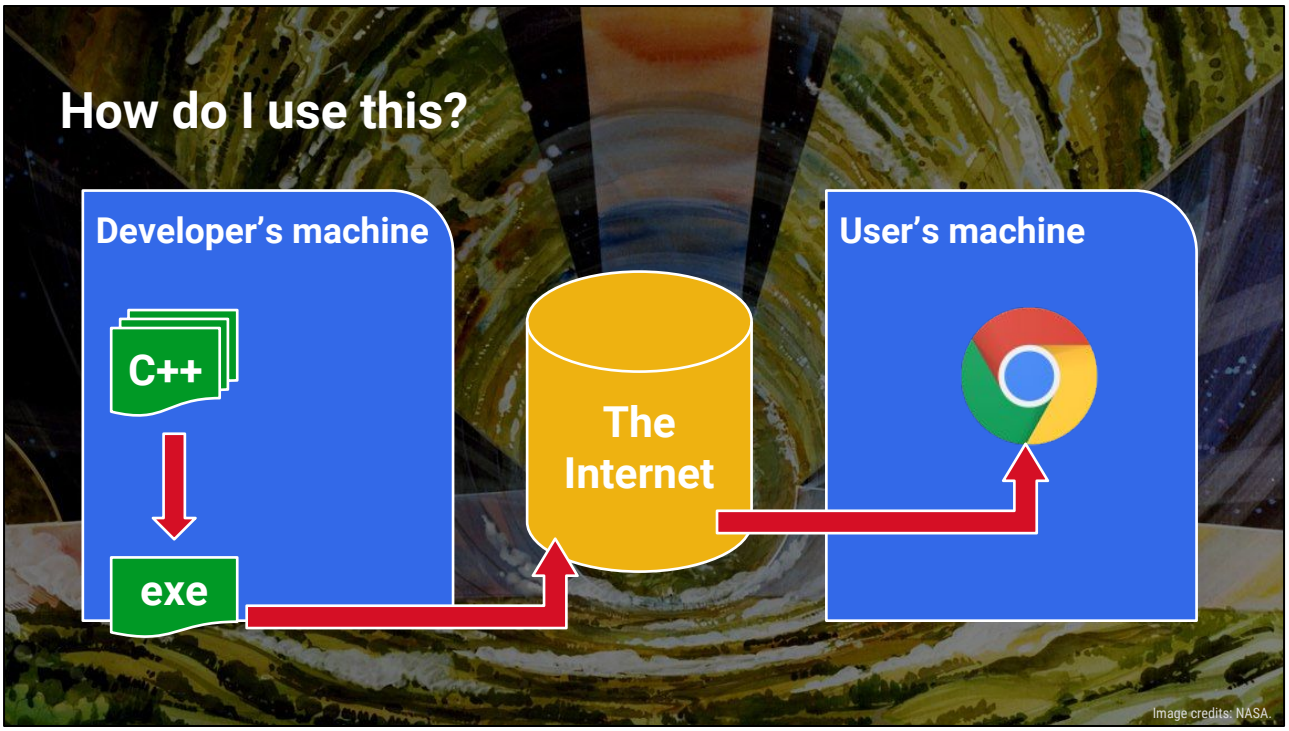
The web can do more than HTML, CSS and JavaScript! It often feels like JavaScript is a hammer and everything looks like a nail.

Ideally, the latest and greatest C++ with full concurrency and parallelism would Just Work. There's *a lot* of existing C/C++ code, it would be great to reuse it. Performance and battery life matter, and C++ can deliver it.

Quick background on NaCl/PNaCl, Emscripten, asm.js, and the upcoming WebAssembly:

- Multiple technologies predate this, including NPAPI and ActiveX.
- NaCl started around the same time Chrome did, but was architecture-specific. Solid sandbox though!
- PNaCl started two years later, to fix the portability problem. Based on LLVM, but took three years to be stable.
- In the meantime, Emscripten came out, compiling to JavaScript.
- asm.js added a clever type system on top of this.
- Now, all four major browser vendors are working together to create a first-class platform: WebAssembly. Don't burden JavaScript, don't leak its flaws to WebAssembly, and make it much more capable over time. This is supplementing the web!

Image credits NASA: http://settlement.arc.nasa.gov/70sArtHiRes/70sArt/art.html



Compile on your machine as you usually would, this creates an executable. Host on the Internet. User downloads content. Content executes.
Seems simple!

Except the user's machine may be very different from the one you compiled the code with, and the platform presents itself as a separate operating system than what you're used to.

And the user should trust any random binary from the internet?

Chrome logo: http://www.google.com/press/images.html
Image credits NASA: http://settlement.arc.nasa.gov/70sArtHiRes/70sArt/art.html

To a certain degree, developers don't really care about user's security: they want their application to be secure, but they have no malicious intent toward the user.

Unfortunately that type of trust doesn't work. The internet is dark and full of terror (but I'm sure you're all nice people with good intentions). Browser developers have to fight for the user (like Tron).

This doesn't mean that your application's own security doesn't matter! It just means that it's not the problem that browser vendors are tackling. We want the toolchains we create to allow you to build something secure, but we're not forcing security (and its associated costs) upon your application beyond security for the user's sake.

Image credits: https://commons.wikimedia.org/wiki/File:Tronguy.jpg

When designing for security, always ask yourself: what's the attack surface?
What are browsers defending against?

We have no clue what's on the internet, and can't even see what attackers are trying out (browsers don't run on our machines).

Browsers have an accretion of languages, file formats, and APIs. Not always co-designed. Often evolving past their original intent.
And here I am, talking about adding some more.

A few attack surfaces:

- The thing that takes the input: parsers.
- The thing that does something with the input: compilers.
- The thing that checks the thing: validator.
- The thing that got generated: your compiled code.
- The APIs used by the thing: GPUs, file systems, ...

Image credits NASA: http://settlement.arc.nasa.gov/70sArtHiRes/70sArt/art.html

Sanitizers

If you have a single takeaway from this presentation, let it be

sanitizers

The sanitizers: http://clang.llvm.org/docs/index.html

- AddressSanitizer
- ThreadSanitizer
- MemorySanitizer
- DataFlowSanitizer
- LeakSanitizer
- Control Flow Integrity

But! Using the sanitizers is costly. I use C++ **for speed**.
Fret not! You can just use them in your tests, and since you comprehensively test *everything* then **for sure** you'll catch all the bugs and prevent all exploits.

Fuzzing

Image credits: Scott McCloud, Google.

Fuzzing has been used for quite a while! This monkey drawing was in the original Chrome comic back at launch.
Browsers use custom fuzzers per language / input file type.
Fuzzers generate random inputs, run on a bunch of machines, and hopefully hit assertions, or **trip the sanitizer's checks**! Did I mention *sanitizers*?
Until recently, the state of the art in fuzzing was pretty much monkeys, who you had to train for very specific tasks. It worked, but didn't scale very well. Sanitizers made bug detection much better, but training smart monkey is still a pretty specialized task.

Keep in mind: fuzzing isn't a proof that your code is bug-free! It's just statistically correct, insofar as your fuzzer is intelligent enough to generate interesting inputs for your specific attack surface.

Image credits for Chrome comic: https://www.google.com/googlebooks/chrome/

Fuzzing: Beyond the Monkeys

AFL and libFuzzer

➔ Smart compile-time feedback
➔ Almost no configuration

Image credits: Michal Zalewski.

Fuzzers have recently become much smarter.

- afl: http://lcamtuf.coredump.cx/afl/
- libFuzzer: http://llvm.org/docs/LibFuzzer.html

This animation of AFL in action is a great demonstration of fuzzing an image decoder. AFL uses smart compile-time instrumentation and genetic algorithms to find interesting inputs that tickle the image decoder in interesting ways.

We've been using these tools intensively recently, and they've found plenty of bugs. The current bottleneck: engineers who have time to fix the bugs. A crash in LLVM when parsing invalid C++ code? Meh, developers can self-hack all they want. A crash when parsing JavaScript? Hmm...

See Kostya's talk Beyond Sanitizers for great demos of these sanitizers, and more gory details.

Image credits Michal Zalewski: https://twitter.com/lcamtuf/status/571477625224347648

A Browser's Bag of Tricks

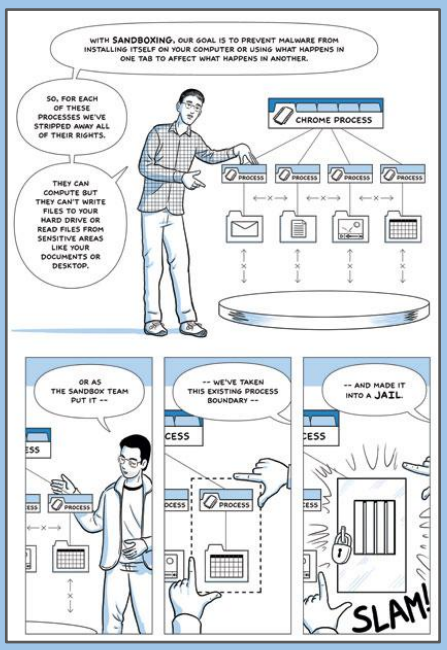


Image credits: Scott McCloud, Google.

I'll concentrate on tooling that's used by browsers. I won't discuss good software development practices, assuming that you already do code reviews and such.

Modern browsers have used clever tricks for quite a while:

- Address Space Layout Randomization, from the OS and hand-baked. Don't make it obvious where things are in the address space.
- Multi processes: an attacker can own one process, but can't affect the other ones without breaking outside of the process.
- Restricting OS syscalls on a per-process basis: if the process gets owned, it still can't do much because it's low-privileged. Defense in depth!
- Put in hard-coded limits for certain things, e.g. sites [can't generate more than a certain amount of code] before we treat them as malicious.
- Use exponential backoff on failure. A crash may be caused by an innocuous bug, by running out of memory... or by an attacker. Some attacks play a game of statistics, throttling mitigates this.
- Collect opt-in anonymous statistics on crashes and API usage. Want to help us capture these issues? You can opt-in! This only catches things at scale, though, not targeted attacks.
    - This is something that most applications can do better than Chrome: monitoring. If you're not monitoring your application, then it's likely broken!
- Frequent release and automatic updates. Something's broken? It'll get fixed

- soon!
- Browsers also ensure different sites can't affect each other through mechanisms such as same-origin policy.
- Google also crawls the web for search, and identifies sites it thinks may be malicious. Safe browsing is then used to alert users in advance if a site may be undesirable.

This is only a subset of what modern browsers do to keep users safe! I'm not even discussing what we do with networking, privacy, or UI. Remember: the weakest link in the chain is often the user. It's the browser's job to help non-technical users understand what's going on, and help them make the right choice.

Keep in mind: browsers are user-mode applications. They don't control the OS they're installed on. They're subject to many other user-space issues such as DLL injections from other installed application! Browsers don't install drivers, and they can't decide to run in a hypervisor. Other types of applications (yours?) may have that luxury and could pull even more clever trick for security's sake. Controlling the OS also gives you great security opportunities, e.g. ChromeOS.

Image credits for Chrome comic: https://www.google.com/googlebooks/chrome/

It honestly feels like attackers have a type Ⅳ bag of holding! What's worse: they sometimes just need one hole to break the browser.

Some examples:

- vtables are writable bug: https://llvm.org/bugs/show_bug.cgi?id=24782
- Type confusion: https://labs.mwrinfosecurity.com/blog/2013/04/19/mwr-labs-pwn2own-2013-write-up---webkit-exploit/
- pwn2own 2015 TOCTOU bug: https://code.google.com/p/chromium/issues/detail?id=468936
- Use after free: https://code.google.com/p/chromium/issues/detail?id=168982
- Integer overflow: https://codereview.chromium.org/61753013/

Pretty much all previous exploits on Chrome can be found in our bug tracker! You can simply search for high-severity security bugs if you're so inclined.

And then there's the JavaScript engine. It's a dynamic programming language, where types change pretty often and the compiler pulls a bunch of tricks **for speed**.
Speaking of speed, it's optimized with a clever Just-in-Time compiler, which supports 9 architectures, and to date **all** browsers basically trust the entire compiler. They are thoroughly tested and fuzzed, but this is hard work!
Here are a few interesting attacks and mitigations: https://www.nccgroup.

trust/us/about-us/resources/jit/

- Write JavaScript that contains 64-bit floating point immediates, cleverly selected. The compiler obligingly embeds these values as-is in executable pages, as immediates. The attacker confused the code, and jumps *into* an immediate, executing it as code. Oops, that was a syscall.
- The compiler maps memory pages as read/write/execute, because it modifies the code often. The attacker confuses the code, and writes over the code instead of inside JavaScript objects.

Sometimes we think we understand a form of exploit, and then things get complicated:

- Return Oriented Programming.
- Blind Return Oriented Programming.
- Return Oriented Programming without Returns.

Chrome has a vulnerability reward program, and that's let to many great reports for those willing to sell us bugs they find. However, it doesn't keep determined malicious attackers at bay.

Image credits: JF Bastien.

Surely we can do better with C++. It's a static language, so none of that runtime compiler business, and we have the sanitizer experience to guide us.

I want to believe that C++ is better!

Image credits: JF Bastien.

NaCl: Simple Validator

```
IP = 0; icount = 0; JumpTargets = { }
  while IP <= TextLimit:
    if inst_is_disallowed(IP):
      error "Disallowed instruction seen"
    StartAddr[icount++] = IP
    if inst_overlaps_block_size(IP):
      error "Block alignment failure"
    if inst_is_indirect_jump_or_call(IP):
      if !is_2_inst_nacl_jmp_idiom(IP)
        or icount < 2
        or Block(StartAddr[icount-2]) != Block(IP):
        error "Bad indirect control transfer"
    else
      JumpTargets = JumpTargets + { IP }
      IP += InstLength(IP)
```

```
for I = 0 to length(StartAddr)-1:
  IP = StartAddr[I]
  if inst_is_direct_jump_or_call(IP):
    T = direct_jump_target(IP)
    if not(T in [0:TextLimit))
      or not(T in JumpTargets):
      error "call/jmp to invalid address"
```

It turns out that NaCl did a pretty great job at this quite a while ago! Don't trust the compiler, rely on a simple validator instead. Verify that code arbitrary code can't do anything unexpected, or read/write anywhere unexpected.

It does this by:

- Being in its own process.
- Forbidding "undesirable" instructions.
- Forcing all jumps to be "bundle" aligned.
- Forbidding instructions from straddling bundle boundaries.
- Make code non-writable.
- Placing "trusted" code into a part of the process that's only reachable through tightly-controlled trampolines, and not directly writable/readable.

You can verify each bundle independently, and if they're all safe then the entire binary is safe.

We've never had to patch NaCl in a stable version of Chrome. It's been used to confuse some APIs, we've tightened it over time (e.g. disallow more instructions), but to our knowledge the sandbox itself was never broken.

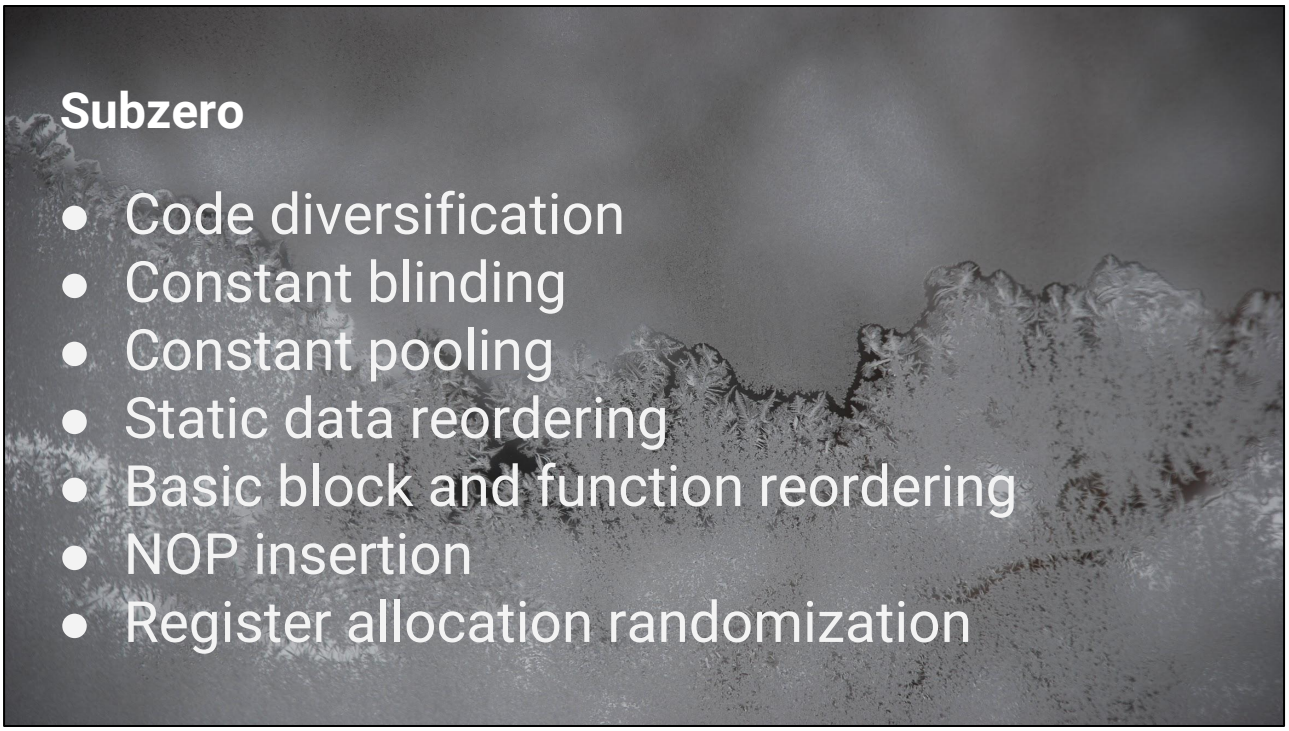
PNaCl uses NaCl as an implementation detail, but works across architectures (x86-32, x86-64, ARM, MIPS). Its translator (from portable executable to architecture-specific

executable) itself is sandboxed! Remember: it's part of the attack surface.

Unfortunately:

- NaCl requires that you use a specialized compiler (forks of GCC or recent LLVM), which aren't upstream. Open source fail :-(
- It adds some runtime overhead (10%-25% geomean on CPU benchmarks).
- It's Chrome only.

Read the full NaCl paper here: http://static.googleusercontent.com/media/research.google.com/en/us/pubs/archive/34913.pdf
Learn about the current sandbox implementations here: https://developer.chrome.com/native-client/reference/sandbox_internals/index



Subzero is the new in-browser compiler for PNaCl. It's doing even more for security, while being smaller and faster than LLVM is:

https://chromium.googlesource.com/native_client/pnacl-subzero/+/master/DESIGN.rst

Similar mitigations can be implemented in LLVM, but at publication time not much of this has made it to tip-of-tree LLVM (yet!). These tricks should all be usable from your code, if you end up having a Just-in-Time compiler which consumes untrusted inputs.

Image credits: JF Bastien.

WebAssembly is still in its early infancy at time of presentation. The high-bits on the design are:

- The C++ program's "heap" is a continuous area of memory. That makes it easy to force all untrusted reads and write to remain in this range.
- The C++ program runs in the same process as the rest of the web page, which reduces the cost of "syscalls".
- Code isn't readable, which means we're exposing something similar to a Harvard architecture (which are wonderful for security). It does imply that some things (such as indirect calls) can end up being more expensive.
- C++ undefined behavior from untrusted code isn't fully undefined: it only leads to limited nondeterminism which can't affect anything outside of the WebAssembly module.

Image credits: JF Bastien.

This was

C++ on the Web

@jfbastien

Thanks!