6.858 Lecture 7
Native Client

What's the goal of this paper?

- At the time, browsers allowed any web page to run only JS (+Flash) code.
- Want to allow web apps to run native (e.g., x86) code on user's machine.
    - Don't want to run complex code on server.
    - Requires lots of server resources, incurs high latency for users.
- Why is this useful?
    - Performance.
    - Languages other than JS.
    - Legacy apps.
- Actually being used in the real world.
    - Ships as part of Google Chrome: the NaCl runtime is a browser extension.
    - Web page can run a NaCl program much like a Flash program.
    - Javascript can interact with the NaCl program by passing messages.
    - NaCl also provides strong sandboxing for some other use cases.
- Core problem: sandboxing x86 code.

Using native client:

- https://developers.google.com/native-client/
- Install browser plug in
- Use Nacl tool change to compile C or C++ program
    - There are restrictions on what system calls you can use
    - Example app: games (don't need much systems support)
    - Special interface to talk to browser (in release called Pepper)
- Make a web page that includes Nacl module:

```
  <embed name="nacl_module"
         id="hello_world"
         width=0 height=0
         src="hello_world.nmf"
         type="application/x-nacl" />
```

- Module is "controled" x86 code.

Quick demo:

```
% urxvt -fn xft:Monospace-20
% export NACL_SDK_ROOT=/home/nickolai/tmp/nacl_sdk/pepper_35
% cd ~/6.858/git/fall14/web/lec/nacl-demo
## this is from NaCl's tutorial part1
% vi hello.cc
% vi index.html
% make
% make serve
```

```
## copy-paste and add --no-dir-check as the error message asks
## visit http://localhost:5103/
## change hello.cc to "memset(buf, 'A', 1024);"
% make
% !python
## visit http://localhost:5103/
## ctrl-shift-J, view console
```

What are some options for safely running x86 code?

Approach 0: trust the code developer.

- ActiveX, browser plug-ins, Java, etc.
- Developer signs code with private key.
- Asks user to decide whether to trust code from some developer.
- Users are bad at making such decisions (e.g., with ActiveX code).
    - Works for known developers (e.g., Windows Update code, signed by MS).
    - Unclear how to answer for unknown web applications (other than "no").
- Native Client's goal is to enforce safety, avoid asking the user.

Approach 1: hardware protection / OS sandboxing.

- Similar plan to some ideas we've already read: OKWS, Capsicum, VMs, ..
- Run untrusted code as a regular user-space program or a separate VM.
- Need to control what system calls the untrusted code can invoke.
    - Linux: seccomp.
    - FreeBSD: Capsicum.
    - MacOSX: Seatbelt.
    - Windows: unclear what options exist.
- Native client uses these techniques, but only as a backup plan.
- Why not rely on OS sandboxing directly?
    - Each OS may impose different, sometimes incompatible requirements.
        - System calls to allocate memory, create threads, etc.
        - Virtual memory layout (fixed-address shared libraries in Windows?).
    - OS kernel vulnerabilities are reasonably common.
        - Allows untrusted code to escape sandbox.
    - Not every OS might have a sufficient sandboxing mechanism.
        - E.g., unclear what to do on Windows, without a special kernel module.
        - Some sandboxing mechanisms require root: don't want to run Chrome as root.
    - Hardware might have vulnerabilities (!).
        - Authors claim some instructions happen to hang the hardware.
        - Would be unfortunate if visiting a web site could hang your computer.

Approach 2: software fault isolation (Native Client's primary sandboxing plan).

- Given an x86 binary to run in Native Client, verify that it's safe.
    - Verification involves checking each instruction in the binary.
    - Some instructions might be always safe: allow.
    - Some instructions might be sometimes safe.
        - Software fault isolation's approach is to require a check before these.
            - Must ensure the check is present at verification time.
        - Another option: insert the check through binary rewriting.
            - Hard to do with x86, but might be more doable with higher-level lang.
    - Some instructions might be not worth making safe: prohibit.
- After verifying, can safely run it in same process as other trusted code.
- Allow the sandbox to call into trusted "service runtime" code. (Figure 2 from paper)

What does safety mean for a Native Client module?

- Goal #1: does not execute any disallowed instructions (e.g., syscall, int).
    - Ensures module does not perform any system calls.
- Goal #2: does not access memory or execute code outside of module boundary.
    - Ensures module does not corrupt service runtime data structures.
    - Ensures module does not jump into service runtime code, ala return-to-libc.
    - As described in paper, module code+data live within [0..256MB) virt addrs.
        - Need not populate entire 256MB of virtual address space.
    - Everything else should be protected from access by the NaCl module.

How to check if the module can execute a disallowed instruction?

- Strawman: scan the executable, look for "int" or "syscall" opcodes.
    - If check passes, can start running code.
    - Of course, need to also mark all code as read-only.
    - And all writable memory as non-executable.
- Complication: x86 has variable-length instructions.
    - "int" and "syscall" instructions are 2 bytes long.
    - Other instructions could be anywhere from 1 to 15 bytes.
- Suppose program's code contains the following bytes:

```
25 CD 80 00 00
```

- If interpreted as an instruction starting from 25, it is a 5-byte instr:

```
AND %eax, $0x000080cd
```

- But if interpreted starting from CD, it's a 2-byte instr:

```
INT $0x80   # Linux syscall
```

- Could try looking for disallowed instructions at every offset..
    - Likely will generate too many false alarms.
    - Real instructions may accidentally have some "disallowed" bytes.

Reliable disassembly.
- Plan: ensure code executes only instructions that verifier knows about.
- How can we guarantee this? Table 1 and Figure 3 in paper.
- Scan forward through all instructions, starting at the beginning.
- If we see a jump instruction, make sure it's jumping to address we saw.
- Easy to ensure for static jumps (constant addr).
- Cannot ensure statically for computed jumps (jump to addr from register)

Computed jumps.
- Idea is to rely on runtime instrumentation: added checks before the jump.
- For computed jump to %eax, NaCl requires the following code:

```
AND $0xffffffe0, %eax
JMP *%eax
```

- This will ensure jumps go to multiples of 32 bytes.
- NaCl also requires that no instructions span a 32-byte boundary.
- Compiler's job is to ensure both of these rules.
    - Replace every computed jump with the two-instruction sequence above.
    - Add NOP instructions if some other instruction might span 32-byte boundary.
    - Add NOPs to pad to 32-byte multiple if next instr is a computed jump target.
    - Always possible because NOP instruction is just one byte.
- Verifier's job is to check these rules.
    - During disassembly, make sure no instruction spans a 32-byte boundary.
    - For computed jumps, ensure it's in a two-instruction sequence as above.
- What will this guarantee?
    - Verifier checked all instructions starting at 32-byte-multiple addresses.
    - Computed jumps can only go to 32-byte-multiple addresses.
- What prevents the module from jumping past the AND, directly to the JMP?
    - Pseudo-instruction.
- How does NaCl deal with RET instructions?
    - Prohibited -- effectively a computed jump, with address stored on stack.
    - Instead, compiler must generate explicit POP + computed jump code.

Why are the rules from Table 1 necessary?

- C1: executable code in memory is not writable.
- C2: binary is statically linked at zero, code starts at 64K.
- C3: all computed jumps use the two-instruction sequence above.
- C4: binary is padded to a page boundary with one or more HLT instruction.
- C5: no instructions, or our special two-instruction pair, can span 32 bytes.
- C6/C7: all jump targets reachable by fall-through disassembly from start.

Homework Q: what happens if verifier gets some instruction length wrong?

How to prevent NaCl module from jumping to 32-byte multiple outside its code?
- Could use additional checks in the computed-jump sequence.
- E.g.:

```
    AND $0x0fffffe0, %eax
    JMP *%eax
```

Why don't they use this approach?
- Longer instruction sequence for computed jumps.
- Their sequence is 3+2=5 bytes, above sequence is 5+2=7 bytes.
- An alternative solution is pretty easy: segmentation

Segmentation.
- x86 hardware provides "segments".
- Each memory access is with respect to some "segment".
    - Segment specifies base + size.
- Segments are specified by a segment selector: ptr into a segment table.

```
%cs, %ds, %ss, %es, %fs, %gs
```

    - Each instruction can specify what segment to use for accessing memory.
    - Code always fetched using the %cs segment.
- Translation: (segment selector, addr) -> (segbase + addr % segsize).
- Typically, all segments have base=0, size=max, so segmentation is a no-op.
- Can change segments: in Linux, modify_ldt() system call.
- Can change segment selectors: just "MOV %ds", etc.

Limiting code/data to module's size.
- Add a new segment with offset=0, size=256MB.
- Set all segment selectors to that segment.
- Modify verifier to reject any instructions that change segment selectors.
- Ensures all code and data accesses will be within [0..256MB).
- (NaCl actually seems to limit the code segment to the text section size.)

What would be required to run Native Client on a system without segmentation?
- For example, AMD/Intel decided to drop segment limits in their 64-bit CPUs.

- One practical possibility: run in 32-bit mode.
    - AMD/Intel CPUs still support segment limits in 32-bit mode.
    - Can run in 32-bit mode even on a 64-bit OS.
- Would have to change the computed-jump code to limit target to 256MB.
- Would have to add runtime instrumentation to each memory read/write.
- See the paper in additional references below for more details.

Why doesn't Native Client support exceptions for modules?

- What if module triggers hardware exception: null ptr, divide-by-zero, etc.
- OS kernel needs to deliver exception (as a signal) to process.
- But Native Client runs with an unusual stack pointer/segment selector.
- Some OS kernels refuse to deliver signals in this situation.
- NaCl's solution is to prohibit hardware exceptions altogether.
- Language-level exceptions (e.g., C++) do not involve hardware: no problem

What would happen if the NaCl module had a buffer overflow?

- Any computed call (function pointer, return address) has to use 2-instr jump.
- As a result, can only jump to validated code in the module's region.
- Buffer overflows might allow attacker to take over module.
- However, can't escape NaCl's sandbox.

Limitations of the original NaCl design?

- Static code: no JIT, no shared libraries.
- Dynamic code supported in recent versions (see additional refs at the end).

Invoking trusted code from sandbox.

- Short code sequences that transition to/from sandbox located in [4KB..64KB).
- Trampoline undoes the sandbox, enters trusted code.
    - Starts at a 32-byte multiple boundary.
    - Loads unlimited segment into %cs, %ds segment selectors.
    - Jumps to trusted code that lives above 256MB.
    - Slightly tricky: must ensure trampoline fits in 32 bytes.
    - (Otherwise, module could jump into middle of trampoline code..)
    - Trusted code first switches to a different stack: why?
    - Subsequently, trusted code has to re-load other segment selectors.
- Springboard (re-)enters the sandbox on return or initial start.
    - Re-set segment selectors, jump to a particular address in NaCl module.
    - Springboard slots (32-byte multiples) start with HLT.
    - Prevents computed jumps into springboard by module code.

What's provided by the service runtime? NaCl's "system call" equivalent.

- Memory allocation: sbrk/mmap.
- Thread operations: create, etc.
- IPC: initially with Javascript code on page that started this NaCl program.

- Browser interface via NPAPI: DOM access, open URLs, user input, ..
- No networking: can use Javascript to access network according to SOP.

How secure is Native Client?

- List of attack surfaces: start of section 2.3.
- Inner sandbox: validator has to be correct (had some tricky bugs!).
- Outer sandbox: OS-dependent plan.
  - On Linux, probably seccomp.
  - On FreeBSD (if NaCl supported it), Capsicum would make sense.
- Why the outer sandbox?
  - Possible bugs in the inner sandbox.
- What could an adversary do if they compromise the inner sandbox?
  - Exploit CPU bugs.
  - Exploit OS kernel bugs.
  - Exploit bugs in other processes communicating with the sandbox proc.
- Service runtime: initial loader, runtime trampoline interfaces.
- IMC interface + NPAPI: complex code, can (and did) have bugs.

How well does it perform?

- CPU overhead seems to be dominated by NaCl's code alignment requirements.
  - Larger instruction cache footprint.
  - But for some applications, NaCl's alignment works better than gcc's.
- Minimal overhead for added checks on computed jumps.
- Call-into-service-runtime performance seems comparable to Linux syscalls.

How hard is it to port code to NaCl?

- For computational things, seems straightforward: 20 LoC change for H.264.
- For code that interacts with system (syscalls, etc), need to change them.
  - E.g., Bullet physics simulator (section 4.4).

Additional references.

- Native Client for 64-bit x86 and for ARM.
  - http://static.usenix.org/events/sec10/tech/full_papers/Sehr.pdf
- Native Client for runtime-generated code (JIT).
  - http://research.google.com/pubs/archive/37204.pdf
- Native Client without hardware dependence.
  - http://css.csail.mit.edu/6.858/2012/readings/pnacl.pdf
- Other software fault isolation systems w/ fine-grained memory access control.
  - http://css.csail.mit.edu/6.858/2012/readings/xfi.pdf
  - http://research.microsoft.com/pubs/101332/bgii-sosp.pdf
- Formally verifying the validator.
  - http://www.cse.lehigh.edu/~gtan/paper/rocksalt.pdf