Native Client: A Sandbox for portable, untrusted x86 Native Code

B. Yee, D. Sehr, G. Dardyj, J.B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar

Google Inc.

Chinmay Nirkhe
nirkhe@cs.berkeley.edu
Python programs

Use different systems!
Secure execution of native code in the web browser
Secure execution of native code in the web browser

- Existing software components
- Legacy desktop applications
- Heavy computation in enterprise applications
- Multimedia applications
- Games
- Any application that requires acceleration
Access to the Native client, gives the ability to provide code that is optimized for the user’s system.
Oblivious of the specifics of the client

Generic browser-based applications cannot access the clients full computational potential without hindering security.
Oblivious of the specifics of the client, plugin-based applications cannot access the client's full computational potential without hindering security.
Oblivious of the specifics of the client, generic browser-based applications cannot access the client's full computational potential without hindering security.

Can you trust me?

Yeah, Ok. Here are my specs.
The Challenge

Provide browser-based applications the computational performance of native applications without compromising safety.
What happens in the Sandbox, stays in the Sandbox.
Some code to turn into binary

NaCl toolchain

Testing

Generate Hardware Optimized Sandboxes

Test binary code for security loopholes

Run on respective clients

naCl executable
The onerous of developing for different systems is now on Native Client and no longer on the developer.
Browser

User Interface
HTML and Javascript

Inter-module communication

NaCl Container

nexes (ex. Imglib.nexe)

Trusted

Untrusted
DANGER

DETAILS AHEAD

STOP PAYING ATTENTION NOW IF SUMMARY WAS SUFFICIENT
I've got a great idea for school tomorrow.

I cut a ping-pong ball in half, and now I'm drawing dots on each end.

I'll just put one over each eye, and it will look like I'm really paying attention.

Or will I look too interested?

I doubt it. I'm over here.
But how can I trust the binary code that the server wants to run on my client?

Don’t trust me. Trust NaCl. I’m totally sandboxed!

Google claims that NaCl can handle

- Untrusted modules from any website
- Unintended side effects outside the sandbox
- Handle allocation of memory and additionally spawned threads
- Handle any race condition attacks
Inner Sandbox

Outer Sandbox

- Handles the same security threats as standard Sandbox used by OS
- Additional level of security not strictly necessary

- Static analysis to discover security defects in x86 code
- Disallow self-modifying code and overlapping instructions
- Provide reliable disassembly
Software Fault Isolation

Looks safe to me!
ALU

Moves between registers (in sandbox)

UNSAFE

- instrumentation
- memory access
- jumps/returns

SAFE

- prohibited
- system call
- interrupts

halt
Disallowed instructions (some more examples)

• `syscall` and `int`: untrusted code should not invoke the OS directly
• All instructions that modify the x86 state (ex. `lds`, far calls, etc.)
• `ret`. Returns are implemented with a sandboxing sequence that ends with an indirect jump
Browser

User Interface
HTML and Javascript

Untrusted

Inter-module communication

NaCl Container

nex (ex. Imglib.nexe)

Untrusted

NaCl Container

Allocmem.nexe

Trusted

Trusted

Trusted

Other trusted modules: threads, messaging, ...
2 Parts to Security

No disallowed instructions

All code and data access is in the module bounds

NOT | JUMP | ADD | SUB | XOR | JUMP | ...

0

Variable Instruction Length!

25 CD 80 00 00

Example credit to N. Zeldovich
2 Parts to Security

No disallowed instructions

All code and data access is in the module bounds

**Reliable Disassembly**

Variable Instruction Length!

```
25  CD  80  00  00
```

int $ 0x80

and %eax, 0x000008CD

Example credit to N. Zeldovich
NaCl has a set of alignment and structural rules such that the native code module can be disassembled such that **all** reachable instructions are identified in disassembly.
Reliable Dissassembly

- Check every mult. of 32 is a valid instruction
- Systematically check instructions
- \textbf{JMP}: check target was seen

\begin{verbatim}
JMP %eax
AND $0xffffffffe0, %eax
JMP %eax
\end{verbatim}
Reliable Dissassembly

- Check every multiple of 32 is a valid instruction
- Systematically check instructions
- `JMP`: check target was seen

\[
\text{NaCl Pseudoinstruction}
\]

\[
\text{JMP \%eax}
\]
Constraints for NaCl binaries

1. Once loaded into the memory, the binary is not writable, enforced by OS-level protection mechanisms during execution.
2. The binary is statically linked at a start address of zero, with the first byte of text at 64K.
3. All indirect control transfers use a `nacljmp` pseudo-instruction.
4. The binary is padded up to the nearest page with at least one `hlt` instruction (0xf4).
5. The binary contains no instructions or pseudo-instructions overlapping a 32-byte boundary.
6. All `valid` instruction addresses are reachable by a fall-through disassembly that starts at the load (base) address.
7. All direct control transfers target valid instructions.
NaCl has a very large and complex architecture...

- Modifier Compiler Toolchain
- Secure ELF Loader
- Disassembler and code verification
- Service Runtime
- Inner and outer sandboxes
- IMC (Inter-module communication)
  - SRPC (Simple Remote Procedure Call)
  - PPAPI (Pepper)
- Trampolines and Springboards
- ...
Secure execution of native code in the web browser