ISOLATION

Running untrusted code in a trusted environment

Setting
Possibly with multiple tenants
OS: users / processes
Browser: webpages / browser extensions
Cloud: virtual machines (VMs)

Threat model
Execution begins in the trusted environment
Attacker can provide arbitrary code and data
Attacker’s goal is to run arbitrary code or exfiltrate data

Security goal
Restrict the set of actions that an attacker can make
The Security Architecture of the Chromium Browser

Adam Barth
UC Berkeley
Charles-Reed
University of Washington
Collin Jackson
Stanford University
Google Chrome Team
Google Inc.

ABSTRACT

Most current web browsers employ a modular architecture that consists of "the core" and "the web" into a single protection domain. An attacker who exploits an exploitable code execution vulnerability in such a browser can sense sensitive data or install malware. In this paper, we present the security architecture of Chromium, the open-source browser upon which Google Chrome is built. Chromium has two modules in separate protection domains: a browser kernel and the operating system. This architecture results in high security with a modular architecture and rich security with the KDE modules.

1. INTRODUCTION

In the last seven years, the web has evolved to become a reliable platform for applications. However, modern web browsers still use the original modular architecture introduced by Netscape in 1998. A modular architecture has many advantages for web applications, including high user sensitivity to the system, and the ability to handle web applications efficiently. However, it also has some disadvantages that should be addressed.

In this paper, we present and evaluate the security architecture of Chromium, the open-source browser upon which Google Chrome is built. Chromium uses a modular architecture, allowing the browser to separate important security concerns in different modules. Chromium's security architecture includes secure kernel, the operating system, and a sandbox for web applications. These features make Chromium a secure and reliable web browser.

There have been a number of research proposals for modular browsers architectures [1], [2], [5], [7] that seek to improve the protection domains. Like Chromium’s architecture, these proposals trade off complexity with usability. However, Chromium’s architecture is simple and can achieve similar goals without sacrificing usability. Chromium’s architecture is based on a modular browser architecture.

In this paper, we describe Chromium’s architecture and how it addresses the security problems of the modern web browser. We also describe the design and implementation of the Chromium’s security architecture and the security features provided by Chromium.

There have been a number of research proposals for modular browsers architectures [1], [2], [5], [7] that seek to improve the protection domains. Like Chromium’s architecture, these proposals trade off complexity with usability. Chromium’s architecture is simple and can achieve similar goals without sacrificing usability. Chromium’s architecture is based on a modular browser architecture.

In this paper, we describe Chromium’s architecture and how it addresses the security problems of the modern web browser. We also describe the design and implementation of the Chromium’s security architecture and the security features provided by Chromium.
What have I done to deserve this?
SANDBOXES

Execution environment that restricts what an application running in it can do

NaCl’s restrictions
Takes arbitrary x86, runs it in a sandbox in a browser
Restrict applications to using a narrow API
Data integrity: No reads/writes outside of sandbox
No unsafe instructions
CFI

Chromium’s restrictions
Runs each webpage’s rendering engine in a sandbox
Restrict rendering engines to a narrow “kernel” API
Data integrity: No reads/writes outside of sandbox (incl. the desktop and clipboard)
## NACL Constraints

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>Once loaded into the memory, the binary is not writable, enforced by OS-level protection mechanisms during execution.</td>
</tr>
<tr>
<td>C2</td>
<td>The binary is statically linked at a start address of zero, with the first byte of text at 64K.</td>
</tr>
<tr>
<td>C3</td>
<td>All indirect control transfers use a <code>nacljmp</code> pseudo-instruction (defined below).</td>
</tr>
<tr>
<td>C4</td>
<td>The binary is padded up to the nearest page with at least one <code>hlt</code> instruction (0xf4).</td>
</tr>
<tr>
<td>C5</td>
<td>The binary contains no instructions or pseudo-instructions overlapping a 32-byte boundary.</td>
</tr>
<tr>
<td>C6</td>
<td>All <em>valid</em> instruction addresses are reachable by a fall-through disassembly that starts at the load (base) address.</td>
</tr>
<tr>
<td>C7</td>
<td>All direct control transfers target valid instructions.</td>
</tr>
</tbody>
</table>

Applied to all untrusted binaries
## NAACL CONSTRAINTS

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>Once loaded into the memory, the binary is not writable, enforced by OS-level protection mechanisms during execution.</td>
</tr>
<tr>
<td>C2</td>
<td>The binary is statically linked at a start address of zero, with the first byte of text at 64K.</td>
</tr>
<tr>
<td>C3</td>
<td>All indirect control transfers use a <code>nacljmp</code> pseudo-instruction (defined below).</td>
</tr>
<tr>
<td>C4</td>
<td>The binary is padded up to the nearest page with at least one <code>hlt</code> instruction (0xf4).</td>
</tr>
<tr>
<td>C5</td>
<td>The binary contains no instructions or pseudo-instructions overlapping a 32-byte boundary.</td>
</tr>
<tr>
<td>C6</td>
<td>All valid instruction addresses are reachable by a fall-through disassembly that starts at the load (base) address.</td>
</tr>
<tr>
<td>C7</td>
<td>All direct control transfers target valid instructions.</td>
</tr>
</tbody>
</table>

### What if we *didn’t* have this?

Attacker could overwrite the binary with code (e.g., as a result of a `wget`)

NaCl would have to statically analyze that new code

### What if we *only* had this?

Load binary with invalid instructions

ROP to make the binary writable
What if we *didn’t* have this? Would render C5, C6, C7 useless

⇒ Could not determine control transfer targets

What if we *only* had this? Alone, it is not checking for or preventing anything

---

**NACL CONSTRAINTS**

<table>
<thead>
<tr>
<th>C1</th>
<th>Once loaded into the memory, the binary is not writable, enforced by OS-level protection mechanisms during execution.</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2</td>
<td>The binary is statically linked at a start address of zero, with the first byte of text at 64K.</td>
</tr>
<tr>
<td>C3</td>
<td>All indirect control transfers use a <em>nacljmp</em> pseudo-instruction (defined below).</td>
</tr>
<tr>
<td>C4</td>
<td>The binary is padded up to the nearest page with at least one <em>hlt</em> instruction (0xf4).</td>
</tr>
<tr>
<td>C5</td>
<td>The binary contains no instructions or pseudo-instructions overlapping a 32-byte boundary.</td>
</tr>
<tr>
<td>C6</td>
<td>All <em>valid</em> instruction addresses are reachable by a fall-through disassembly that starts at the load (base) address.</td>
</tr>
<tr>
<td>C7</td>
<td>All direct control transfers target valid instructions.</td>
</tr>
</tbody>
</table>
What if we didn’t have this?  Attacker could potentially jump anywhere ROP, code injection

What if we only had this?  C1 necessary; C2 ensures these are instructions C7 ensures that what it’s jumping to is valid

---

<table>
<thead>
<tr>
<th>C1</th>
<th>Once loaded into the memory, the binary is not writable, enforced by OS-level protection mechanisms during execution.</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2</td>
<td>The binary is statically linked at a start address of zero, with the first byte of text at 64K.</td>
</tr>
<tr>
<td>C3</td>
<td>All indirect control transfers use a nacljmp pseudo-instruction (defined below).</td>
</tr>
<tr>
<td>C4</td>
<td>The binary is padded up to the nearest page with at least one hlt instruction (0xf4).</td>
</tr>
<tr>
<td>C5</td>
<td>The binary contains no instructions or pseudo-instructions overlapping a 32-byte boundary.</td>
</tr>
<tr>
<td>C6</td>
<td>All valid instruction addresses are reachable by a fall-through disassembly that starts at the load (base) address.</td>
</tr>
<tr>
<td>C7</td>
<td>All direct control transfers target valid instructions.</td>
</tr>
</tbody>
</table>

---

```
nacljmp (SFI)

jmp %eax

First byte is 64K (C2)

and %eax, 0xfffffffffe0

jmp (%eax)
```

## NACL Constraints

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>Once loaded into the memory, the binary is not writable, enforced by OS-level protection mechanisms during execution.</td>
</tr>
<tr>
<td>C2</td>
<td>The binary is statically linked at a start address of zero, with the first byte of text at 64K.</td>
</tr>
<tr>
<td>C3</td>
<td>All indirect control transfers use a <code>nacljmp</code> pseudo-instruction (defined below).</td>
</tr>
<tr>
<td>C4</td>
<td>The binary is padded up to the nearest page with at least one <code>hlt</code> instruction (0xf4).</td>
</tr>
<tr>
<td>C5</td>
<td>The binary contains no instructions or pseudo-instructions overlapping a 32-byte boundary.</td>
</tr>
<tr>
<td>C6</td>
<td>All valid instruction addresses are reachable by a fall-through disassembly that starts at the load (base) address.</td>
</tr>
<tr>
<td>C7</td>
<td>All direct control transfers target valid instructions.</td>
</tr>
</tbody>
</table>

**What if we didn’t have this?** Execution would continue beyond the executable itself. Could start to run data.

**What if we only had this?** Provides no guarantees about what’s in the code itself.
**NACL CONSTRAINTS**

What if we *didn’t* have this?  
Would render nacljmp useless  
⇒ Wouldn’t know what exactly we’re jumping to

What if we *only* had this?  
Provides no guarantees about what we are jumping to
### NACL CONSTRAINTS

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>Once loaded into the memory, the binary is not writable, enforced by OS-level protection mechanisms during execution.</td>
</tr>
<tr>
<td>C2</td>
<td>The binary is statically linked at a start address of zero, with the first byte of text at 64K.</td>
</tr>
<tr>
<td>C3</td>
<td>All indirect control transfers use a <code>nacljmp</code> pseudo-instruction (defined below).</td>
</tr>
<tr>
<td>C4</td>
<td>The binary is padded up to the nearest page with at least one <code>hlt</code> instruction (0xf4).</td>
</tr>
<tr>
<td>C5</td>
<td>The binary contains no instructions or pseudo-instructions overlapping a 32-byte boundary.</td>
</tr>
<tr>
<td>C6</td>
<td>All <em>valid</em> instruction addresses are reachable by a fall-through disassembly that starts at the load (base) address.</td>
</tr>
<tr>
<td>C7</td>
<td>All direct control transfers target valid instructions.</td>
</tr>
</tbody>
</table>

**What if we didn’t have this?**

Could not perform disassembly

⇒ Could not infer what instructions are called

**What if we only had this?**

C1 still breaks it

Doesn’t say you can’t also hit invalid instructions
# NACL Constraints

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>Once loaded into the memory, the binary is not writable, enforced by OS-level protection mechanisms during execution.</td>
</tr>
<tr>
<td>C2</td>
<td>The binary is statically linked at a start address of zero, with the first byte of text at 64K.</td>
</tr>
<tr>
<td>C3</td>
<td>All indirect control transfers use a nacljmp pseudo-instruction (defined below).</td>
</tr>
<tr>
<td>C4</td>
<td>The binary is padded up to the nearest page with at least one hlt instruction (0xf4).</td>
</tr>
<tr>
<td>C5</td>
<td>The binary contains no instructions or pseudo-instructions overlapping a 32-byte boundary.</td>
</tr>
<tr>
<td>C6</td>
<td>All valid instruction addresses are reachable by a fall-through disassembly that starts at the load (base) address.</td>
</tr>
<tr>
<td>C7</td>
<td>All direct control transfers target valid instructions.</td>
</tr>
</tbody>
</table>

What if we *didn’t* have this? 
Invalid instructions! 
⇒ Arbitrary syscalls, interrupts, loads, returns, ...

What if we *only* had this? 
C1 still breaks it; C4: could execute beyond the binary 
C2, C3, C5, C6 are needed to get to C7
C2: Known entry point
C7: No invalid instructions
C5: No invalid alignments
C3: Only use nacljmp

Common disassembly techniques

```c
// TextLimit = the upper text address limit
// Block(IP) = 32-byte block containing IP
// StartAddr = list of inst start addresses
// JumpTargets = set of valid jump targets

// Part 1: Build StartAddr and JumpTargets
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_jump_idiom(IP) or icount < 2 or
            Block(StartAddr[icount-2]) != Block(IP):
            error "Bad indirect control transfer"
    else
        // Note that indirect jmps are inside
        // a pseudo-inst and bad jump targets
        JumpTargets = JumpTargets + { IP }
    // Proceed to the fall-through address
    IP += InstLength(IP)

// Part 2: Detect invalid direct transfers
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"
```
**DISASSEMBLY**

Control Flow Integrity for COTS Binaries

Mingwei Zhang and R. Sekar
Stony Brook University
Stony Brook, NY, USA.

Abstract
Control-Flow Integrity (CFI) has been recognized as an important low-level security property. Its enforcement can defeat most injected and existing code attacks, including those based on Return-Oriented Programming (ROP). Previous implementations of CFI have required compiler support or the presence of relocation or debug information in the binary. In contrast, we present a technique for applying CFI to stripped binaries on x86/Linux. Ours is the first work to apply CFI to complex shared libraries such as glibc. Through experimental evaluation, we demonstrate that our CFI implementation is effective against control-flow hijack attacks, and eliminates the vast majority of ROP gadgets. To achieve this result, we have developed robust techniques for disassembly, static analysis, and transformation of large binaries. Our techniques have been tested on over 300MB of binaries (executables and shared libraries).

1 Introduction
Since its introduction by Abadi et. al. [1, 2], Control-Flow Integrity (CFI) has been recognized as an important low-level security property. Unlike address-space randomization [24, 5] and stack cookies [12, 17], CFI’s control-flow hijack defense is not vulnerable to the release of binaries. Indeed, some applications of CFI, such as sandboxing untrusted code, explicitly target binaries. Most existing CFI implementations, including those in Native Client [46], Pittsfeld [27], Control-flow locking [6] and many other works [22, 3, 42, 4, 36] are implemented within compiler tool chains. They rely on information that is available in assembly code or higher levels, but unavailable in COTS binaries. The CFI implementation of Abadi et al [2] relies on relocation information. Although this information is included in Windows libraries that support ASLR, UNIX systems (and specifically, Linux systems) rely on position-independent code for randomization, and hence do not include relocation information in COTS binaries. We therefore develop a new approach for enforcing CFI on COTS binaries without relocation or other high-level information.

Despite operating with less information, the security and performance provided by our approach are comparable to that of the existing CFI implementations. Moreover, our implementation is robust enough to handle complex executables as well as shared libraries. We begin by summarizing our approach and results.

1.1 CFI for COTS Binaries
We present the first practical approach for CFI enforcement that scales to large binaries as well as shared

---

**Linear disassembly**
Start at instruction i
i += inst_len(i)

Leaves gaps if there are variable-length inst’s, data, bad alignment...

**Recursive disassembly**
Set of entry points E
Start at entry point i
if i is a jmp:
   add its target to E
i += inst_len(i)

---

**Goal:** CFI without access to code:
How do you infer the control flow graph?
C2: Known entry point
C7: No invalid instructions
C5: No invalid alignments
C3: Only use nacljmp

**Theorem:** StartAddr contains all addresses that can be reached from an instruction with address in StartAddr.
Theorem: StartAddr contains all addresses that can be reached from an instruction with address in StartAddr.

**Case 1:** IP is reached by falling through from A. This implies that IP is \( \text{InstAddr}(A) + \text{InstLength}(A) \). But this address would have been in \( S \) from part 1 of the construction. Contradiction.

**Case 2:** IP is reached by a direct jump or call from an instruction A in \( S \). Then IP must be in JumpTargets, a condition checked by part 2 of the construction. Observe that JumpTargets is a subset of \( S \) from part 1 of the construction. Therefore IP must be in \( S \). Contradiction.

**Case 3:** IP is reached by an indirect transfer from an instruction at A in \( S \). Since the instruction at A is an indirect call or jump, any execution of A always immediately follows the execution of an and. After the and the computed address is aligned \( 0 \mod 32 \). Since no instruction can straddle a \( 0 \mod 32 \) boundary, every \( 0 \mod 32 \) address in \( \text{[0, TextLimit)} \) must be in \( S \). Hence IP is in \( S \). Contradiction.
ACTUALLY DOING THINGS WITH NACL

C2  The binary is statically linked at a start address of zero, with the first byte of text at 64K.

First 4KB: Unreadable, unwritable (detect NULL pointers)

Remaining 60KB: **trusted** trampoline code (untrusted to trusted) & springboard return (trusted to untrusted)

Ensures we have a **Trusted Compute Base (TCB)** in the malicious binary

Allowed to contain instructions that are forbidden elsewhere

Especially **far call** to enable control transfers between untrusted user code and trusted service runtime

Separation is handled by setting / restoring **segment registers**, which locate the code/text segments
**Inner sandbox**

Untrusted 3rd-party data

Untrusted 3rd-party code

Trampoline

Springboard

**Outer sandbox**

Mediates system calls at the process boundary

**Swap between untrusted & trusted within a process via segment registers**

System calls

**Trusted code**

**Trusted data**
SECURITY DESIGN PRINCIPLES

Defense in depth
SECCOMP-BPF

- Linux system call enabled since 2.6.12 (2005)
  - Affected process can subsequently **only perform read, write, exit, and sigreturn system calls**
    - No support for open call: Can only use already-open file descriptors
  - **Isolates a process by limiting possible interactions**

- Follow-on work produced **seccomp-bpf**
  - Limit process to policy-specific set of system calls, subject to a policy handled by the kernel
    - Policy akin to Berkeley Packet Filters (BPF)
  - **Used by Chrome, OpenSSH, vsftpd, and others**
TODAY'S PAPERS

Native Client: A Sandbox for Portable, Untrusted x86 Native Code
Benjamin Yee, David Sia, Gregory Darley, I. Bradley Chen, Robert Math, Tavis Ormandy, Shlomi Dorchin, Neha Rimala, and Nicholas Filliger
Google Inc.

Abstract
This paper describes the design, implementation and evaluation of Native Client, a sandbox for untrusted x86 native code. Native Client aims to give browser-based applications the computational performance of native applications without the performance benefits and security advantages of Native Client. Native Client provides operating system separation for binary code, while supporting performance-optimization techniques such as thread support, instruction set extensions such as SSE, and use of compiler intrinsics andheatfeld assembly. We evaluate these techniques in an open application that encompasses community review and feedback.

1. Introduction
As an application platform, the modern web browser brings together a remarkable confluence of technologies, including seamless access to Internet resources, high-performance languages such as JavaScript, and the richness of the Document Object Model (DOM) [46] for graphics presentation and user interaction. While these strengths help the browser to serve as a target for new software development, it remains handicapped in a critical dimension: computational performance. Thanks to Moore’s Law, the web is clearly not the same as it was a few years ago: the current generation of personal computers can now handle the most demanding web pages in an instant. Unfortunately, the current generation of web browsers is not designed for performance:

The main contributions of this work are as follows:

1. An architecture for native code execution for untrusted x86 native code.
2. An open source implementation of the native code module.
3. A new set of technologies for improving the native code execution environment.
4. A new set of technologies for improving the native code execution environment.
5. A new set of technologies for improving the native code execution environment.

1. Introduction
In this paper, we present Native Client, a new application platform for web browsers. Native Client provides a new model for web application execution, allowing web applications to run as efficiently as native applications.

The security architecture of Chromium Browser

ABSTRACT
The Chromium browser is a new browser that can run untrusted X86 native code. Chromium uses several techniques to prevent execution of malicious native code, such as a new execution architecture, an access control mechanism that prevents execution of malicious native code, and a new security model that prevents execution of malicious native code.

In this paper, we present the security architecture of Chromium. The security architecture of Chromium includes several key components: a new execution architecture, a new access control mechanism, and a new security model.

The security architecture of Chromium includes several key components:

1. A new execution architecture.
2. A new access control mechanism.
3. A new security model.

1. Introduction
In this paper, we present the security architecture of Chromium. The security architecture of Chromium includes several key components: a new execution architecture, a new access control mechanism, and a new security model.

The security architecture of Chromium includes several key components:

1. A new execution architecture.
2. A new access control mechanism.
3. A new security model.

1. Introduction
In this paper, we present the security architecture of Chromium. The security architecture of Chromium includes several key components: a new execution architecture, a new access control mechanism, and a new security model.

The security architecture of Chromium includes several key components:

1. A new execution architecture.
2. A new access control mechanism.
3. A new security model.

1. Introduction
In this paper, we present the security architecture of Chromium. The security architecture of Chromium includes several key components: a new execution architecture, a new access control mechanism, and a new security model.
CHROMIUM ARCHITECTURE

Rendering Engine:
Interprets and executes web content
Outputs rendered bitmaps
The website is the “untrusted code”

Goal: Enforce a narrow interface between the two

Browser Kernel:
Stores data (cookies, history, clipboard)
Performs all network operations
CHROMIUM’S SANDBOX

Makes extensive use of the underlying OS’s primitives

1. Restricted security token
   The OS then provides complete mediation on access to “securable objects”
   (Security token set s.t. it fails almost always)

2. Separate desktop
   Avoid Windows API’s lax security checks

3. Windows Job Object
   Can’t fork processes; can’t access clipboard
CHROMIUM’S BROWSER KERNEL INTERFACE

Goal: Do not leak the ability to read or write the user’s file system

1. Restrict rendering
   Rendering engine doesn’t get a window handle
   Instead, draws to an off-screen bitmap
   Browser kernel copies this bitmap to the screen

2. Network & I/O
   Rendering engine requests uploads, downloads, and file access thru BKI

3. Restrict user input
   Rendering engine doesn’t get user input directly
   Instead, browser kernel delivers it via BKI