

ISOLATION DEFENSES

GRAD SEC

OCT 03 2017



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

TODAY'S PAPERS

Will appear in the 2009 IEEE Symposium on Security and Privacy

Native Client: A Sandbox for Portable, Untrusted x86 Native Code

Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth,
Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fallagar
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 compromising safety. Native Client uses software fault isolation and a secure runtime to direct system interaction and side effects through interfaces managed by Native Client. Native Client provides operating system portability for binary code while supporting performance-oriented features generally absent from web application programming environments, such as thread support, instruction set extensions such as SSE, and use of compiler intrinsics and hand-coded assembler. We combine these properties in an open architecture that encourages community review and 3rd-party tools.

1. Introduction

As an application platform, the modern web browser brings together a remarkable combination of resources, including seamless access to Internet resources, high-productivity programming languages such as JavaScript, and the richness of the Document Object Model (DOM) [64] for graphics presentation and user interaction. While these strengths put the browser in the forefront as a target for new application development, it remains handicapped in a critical dimension: computational performance. Thanks to Moore's Law and the zeal with which it is observed by the hardware community, many interesting applications get adequate performance in a browser despite this handicap. But there remains a set of computations that are generally infeasible for browser-based applications due to performance constraints, for example: simulation of Newtonian physics, computational fluid dynamics, and high-resolution scene rendering. The current environment also tends to preclude use of the large bodies of high-quality code developed in languages other than JavaScript.

Modern web browsers provide extension mechanisms such as ActiveX [15] and NPAPI [48] to allow native code to be loaded and run as part of a web application. Such architectures allow plugins to circumvent the security mechanisms otherwise applied to web content, while giving them access to full native performance, perhaps

as a secondary consideration. Given this organization, and the absence of effective technical measures to constrain these plugins, browser applications that wish to use native-code must rely on non-technical measures for security: for example, manual establishment of trust relationships through pop-up dialog boxes, or manual installation of a console application. Historically, these non-technical measures have been inadequate to prevent execution of malicious native code, leading to inconvenience and economic harm [10], [54]. As a consequence we believe there is a prejudice against native code extensions for browser-based applications among experts and distrust among the larger population of computer users.

While acknowledging the insecurity of the current systems for incorporating native-code into web applications, we also observe that there is no fundamental reason why native code should be unsafe. In Native Client, we separate the problem of safe native execution from that of extending trust, allowing each to be managed independently. Conceptually, Native Client is organized in two parts: a constrained execution environment for native code to prevent unintended side effects, and a runtime for hosting these native code extensions through which allowable side effects may occur safely.

The main contributions of this work are:

- an infrastructure for OS and browser-portable sandboxed x86 binary modules,
- support for advanced performance capabilities such as threads, SSE instructions [52], compiler intrinsics and hand-coded assembler,
- an open system designed for easy retargeting of new compilers and languages, and
- refinements to CISC software fault isolation, using x86 segments for improved simplicity and reduced overhead.

We combine these features in an infrastructure that supports safe side effects and local communication. Overall, Native Client provides sandboxed execution of native code and portability across operating systems, delivering native code performance for the browser.

The remainder of the paper is organized as follows. Section 1.1 describes our threat model. Section 2 develops some essential concepts for the NaCl¹ system architecture and

¹ We use "NaCl" as an adjective reference to the Native Client system.

The Security Architecture of the Chromium Browser

Adam Barth^{*}
UC Berkeley
Charles Reis^{*}
University of Washington

Collin Jackson^{*}
Stanford University
Google Chrome Team
Google Inc.

ABSTRACT

Most current web browsers employ a monolithic architecture that combines "the user" and "the web" into a single protection domain. An attacker who exploits an arbitrary code execution vulnerability in such a browser can steal sensitive files 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, which interacts with the operating system, and a rendering engine, which runs with restricted privileges in a sandbox. This architecture helps mitigate high severity attacks without sacrificing compatibility with existing web sites. We define a threat model for browser exploits and evaluate how the architecture would have mitigated past vulnerabilities.

1. INTRODUCTION

In the past several years, the web has evolved to become a viable platform for applications. However, most web browsers still use the original monolithic architecture introduced by NCSA Mosaic in 1993. A monolithic browser architecture has many limitations for web applications with substantial client-side code. For example, a crash caused by a web application takes down the user's entire web experience instead of just the web application that misbehaved [21]. From a security point of view, monolithic web browsers run in a single protection domain, allowing an attacker who can exploit an unpatched vulnerability to compromise the entire browser instance and often run arbitrary code on the user's machine with the user's privileges.

In this paper, we present and evaluate the security architecture of Chromium, the open source web browser upon which Google Chrome is built. Chromium uses a modular architecture, akin to privilege separation in SSHD [18]. The browser kernel module acts on behalf of the user, while the rendering engine module acts on behalf of "the web". These modules run in separate protection domains, enforced by a sandbox that reduces the privileges of the rendering engine. Even if an attacker can exploit an unpatched vulnerability in the rendering engine, obtaining the privileges of the main rendering engine, the sandbox helps prevent the attacker from reading or writing the user's file system because the web principal does not have that privilege.

^{*}The authors conducted this work while employed by Google.

There have been a number of research proposals for modular browser architectures [5, 27, 5, 7] that contain multiple protection domains. Like Chromium's architecture, these proposals aim to provide security against an attacker who can exploit an unpatched vulnerability. Unlike Chromium's architecture, these proposals trade off compatibility with existing web sites to provide architectural isolation between web sites or even individual pages. The browser's security policy, known as the "same-origin policy," is complex and can make such fine-grained isolation difficult to achieve without disrupting existing sites. Users, however, demand compatibility because a web browser is only as useful as the sites that it can render. To be successful, a modular browser architecture must support the entire web platform in addition to improving security.

Chromium's architecture isolates the various core parts of a modern browser between the browser kernel and the rendering engine, balancing security, compatibility, and performance. The architecture allocates high-risk components, such as the HTML parser, the JavaScript virtual machine, and the Document Object Model (DOM), to its sandboxed rendering engine. These components are complex and historically have been the source of security vulnerabilities. Running these components in a sandbox helps reduce the severity of unpatched vulnerabilities in their implementation. The browser kernel is responsible for managing persistent resources, such as cookies and the password database, and for interacting with the operating system to receive user input, draw to the screen, and access the network. The architecture is based on two design decisions:

1. The architecture must be compatible with the existing web. Specifically, the security restrictions imposed by the architecture should be transparent to web sites. This design decision greatly limits the landscape of possible architectures but is essential in order for Chromium to be useful as a web browser. For example, the architecture must support uploading files to web sites in order to be compatible with web-based email sites that let users add attachments to emails.
2. The architecture treats the rendering engine as a black box that takes unparsed HTML as input and produces rendered bitmaps as output (see Figure 1). In particular, the architecture relies on the rendering engine alone to implement the same-origin policy. This design decision reduces the complexity of the browser kernel's security monitor because the browser kernel need only enforce coarse-grained security restrictions. For example, the browser kernel grants the ability to upload a

*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

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

Applied to all untrusted binaries

NACL CONSTRAINTS



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

What if we *didn't*
have this?

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

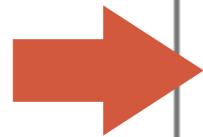
NaCl would have to statically analyze that new code

What if we *only*
had this?

Load binary with invalid instructions

ROP to make the binary writable

NACL CONSTRAINTS



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

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?

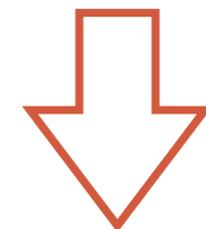
Alone, it is not checking for or preventing anything

NACL CONSTRAINTS

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

`nacljmp (SFI)`

`jmp %eax`



First byte is 64K (C2)

and `%eax, 0xffffffffe0 jmp (%eax)`

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

NACL CONSTRAINTS

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

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

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

What if we *didn't*
have this?

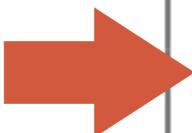
Would render `nacljmp` useless

⇒ 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

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

What if we *didn't*
have this?

Could not perform disassembly

⇒ Could not infer what instructions are called

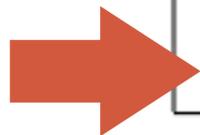
What if we *only*
had this?

C1 still breaks it

Doesn't say you can't also hit invalid instructions

NACL CONSTRAINTS

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



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

NACL VALIDATOR

C2: Known entry point



C7: No invalid instructions



C5: No invalid alignments



C3: Only use nacljmp



Common **disassembly** techniques



```
// 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_jmp_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 re-

fully enforced on binaries. Indeed, some applications of CFI, such as sandboxing untrusted code, explicitly target binaries. Most existing CFI implementations, including those in Native Client [46], Pittsfield [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
 $i += \text{inst_len}(i)$

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
 $i += \text{inst_len}(i)$

Goal: CFI without access to code:

How do you infer the control flow graph?

NACL VALIDATOR

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.

```
// 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"
```

NACL VALIDATOR: PROOF

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 $0 \bmod 32$. Since no instruction can straddle a $0 \bmod 32$ boundary, every $0 \bmod 32$ address in $[0, \text{TextLimit})$ 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

NACL'S SANDBOXES

Inner sandbox

Untrusted
3rd-party
data

Untrusted
3rd-party
code

Trampoline

Springboard

Swap between untrusted & trusted within a process via segment registers

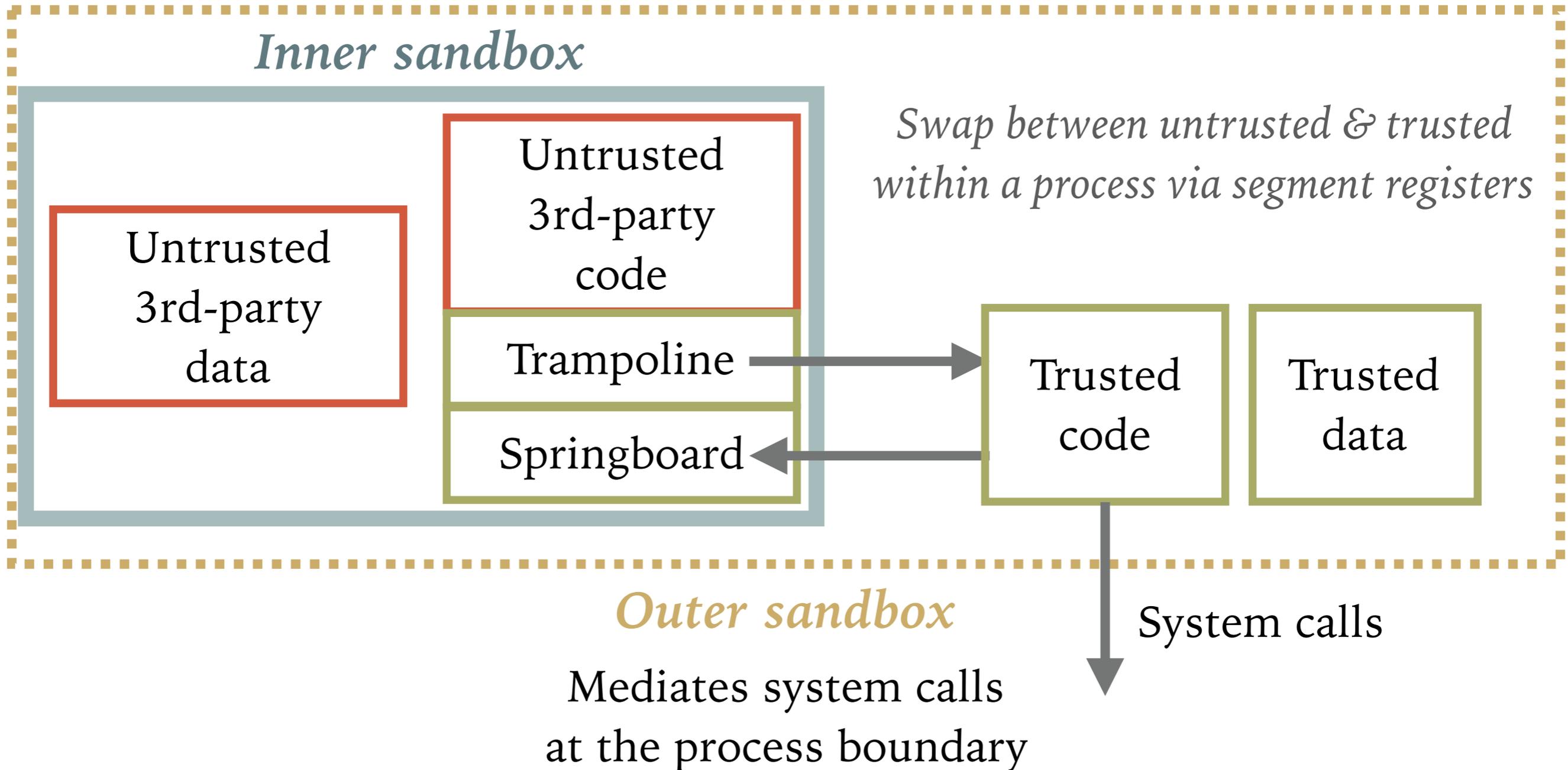
Trusted
code

Trusted
data

Outer sandbox

Mediates system calls
at the process boundary

System calls



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

Will appear in the 2009 IEEE Symposium on Security and Privacy

Native Client: A Sandbox for Portable, Untrusted x86 Native Code

Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth,
Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fallagar
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 compromising safety. Native Client uses software fault isolation and a secure runtime to direct system interaction and side effects through interfaces managed by Native Client. Native Client provides operating system portability for binary code while supporting performance-oriented features generally absent from web application programming environments, such as thread support, instruction set extensions such as SSE, and use of compiler intrinsics and hand-coded assembler. We combine these properties in an open architecture that encourages community review and 3rd-party tools.

1. Introduction

As an application platform, the modern web browser brings together a remarkable combination of resources, including seamless access to Internet resources, high-productivity programming languages such as JavaScript, and the richness of the Document Object Model (DOM) [64] for graphics presentation and user interaction. While these strengths put the browser in the forefront as a target for new application development, it remains handicapped in a critical dimension: computational performance. Thanks to Moore's Law and the zeal with which it is observed by the hardware community, many interesting applications get adequate performance in a browser despite this handicap. But there remains a set of computations that are generally infeasible for browser-based applications due to performance constraints, for example: simulation of Newtonian physics, computational fluid dynamics, and high-resolution scene rendering. The current environment also tends to preclude use of the large bodies of high-quality code developed in languages other than JavaScript.

Modern web browsers provide extension mechanisms such as ActiveX [15] and NPAPI [48] to allow native code to be loaded and run as part of a web application. Such architectures allow plugins to circumvent the security mechanisms otherwise applied to web content, while giving them access to full native performance, perhaps

as a secondary consideration. Given this organization, and the absence of effective technical measures to constrain these plugins, browser applications that wish to use native-code must rely on non-technical measures for security: for example, manual establishment of trust relationships through pop-up dialog boxes, or manual installation of a console application. Historically, these non-technical measures have been inadequate to prevent execution of malicious native code, leading to inconvenience and economic harm [10], [54]. As a consequence we believe there is a prejudice against native code extensions for browser-based applications among experts and distrust among the larger population of computer users.

While acknowledging the insecurity of the current systems for incorporating native-code into web applications, we also observe that there is no fundamental reason why native code should be unsafe. In Native Client, we separate the problem of safe native execution from that of extending trust, allowing each to be managed independently. Conceptually, Native Client is organized in two parts: a constrained execution environment for native code to prevent unintended side effects, and a runtime for hosting these native code extensions through which allowable side effects may occur safely.

The main contributions of this work are:

- an infrastructure for OS and browser-portable sandboxed x86 binary modules,
- support for advanced performance capabilities such as threads, SSE instructions [52], compiler intrinsics and hand-coded assembler,
- an open system designed for easy retargeting of new compilers and languages, and
- refinements to CISC software fault isolation, using x86 segments for improved simplicity and reduced overhead.

We combine these features in an infrastructure that supports safe side effects and local communication. Overall, Native Client provides sandboxed execution of native code and portability across operating systems, delivering native code performance for the browser.

The remainder of the paper is organized as follows. Section 1.1 describes our threat model. Section 2 develops some essential concepts for the NaCl¹ system architecture and

¹ We use "NaCl" as an adjective reference to the Native Client system.

The Security Architecture of the Chromium Browser

Adam Barth^{*}
UC Berkeley
Charles Reis^{*}
University of Washington

Collin Jackson^{*}
Stanford University
Google Chrome Team
Google Inc.

ABSTRACT

Most current web browsers employ a monolithic architecture that combines "the user" and "the web" into a single protection domain. An attacker who exploits an arbitrary code execution vulnerability in such a browser can steal sensitive files 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, which interacts with the operating system, and a rendering engine, which runs with restricted privileges in a sandbox. This architecture helps mitigate high severity attacks without sacrificing compatibility with existing web sites. We define a threat model for browser exploits and evaluate how the architecture would have mitigated past vulnerabilities.

1. INTRODUCTION

In the past several years, the web has evolved to become a viable platform for applications. However, most web browsers still use the original monolithic architecture introduced by NCSA Mosaic in 1993. A monolithic browser architecture has many limitations for web applications with substantial client-side code. For example, a crash caused by a web application takes down the user's entire web experience instead of just the web application that misbehaved [21]. From a security point of view, monolithic web browsers run in a single protection domain, allowing an attacker who can exploit an unpatched vulnerability to compromise the entire browser instance and often run arbitrary code on the user's machine with the user's privileges.

In this paper, we present and evaluate the security architecture of Chromium, the open source web browser upon which Google Chrome is built. Chromium uses a modular architecture, akin to privilege separation in SSHD [18]. The browser kernel module acts on behalf of the user, while the rendering engine module acts on behalf of "the web". These modules run in separate protection domains, enforced by a sandbox that reduces the privileges of the rendering engine. Even if an attacker can exploit an unpatched vulnerability in the rendering engine, obtaining the privileges of the native rendering engine, the sandbox helps prevent the attacker from reading or writing the user's file system because the web principal does not have that privilege.

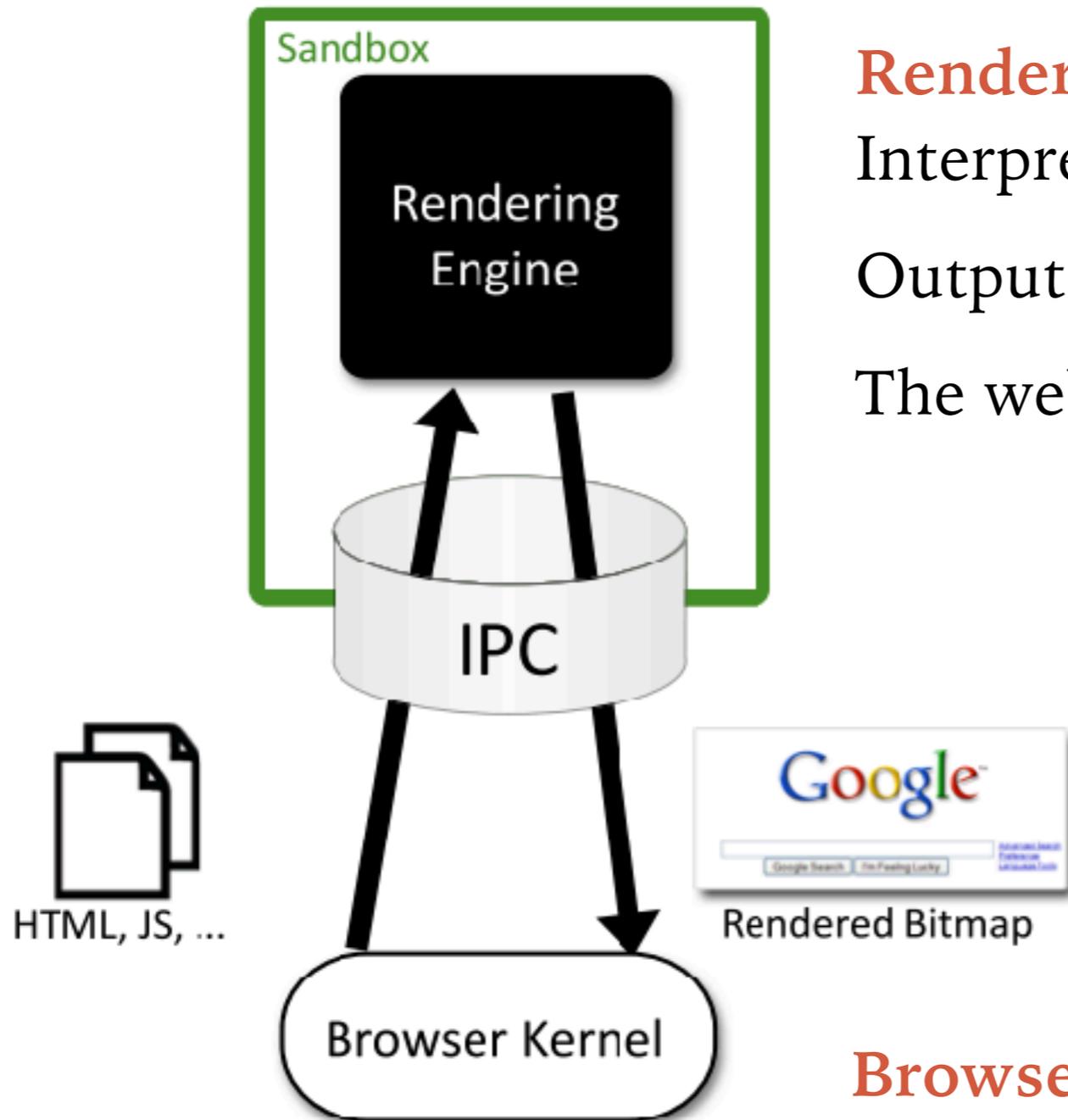
^{*}The authors conducted this work while employed by Google.

There have been a number of research proposals for modular browser architectures [5, 27, 5, 7] that contain multiple protection domains. Like Chromium's architecture, these proposals aim to provide security against an attacker who can exploit an unpatched vulnerability. Unlike Chromium's architecture, these proposals trade off compatibility with existing web sites to provide architectural isolation between web sites or even individual pages. The browser's security policy, known as the "same-origin policy," is complex and can make such fine-grained isolation difficult to achieve without disrupting existing sites. Users, however, demand compatibility because a web browser is only as useful as the sites that it can render. To be successful, a modular browser architecture must support the entire web platform in addition to improving security.

Chromium's architecture isolates the various core parts of a modern browser between the browser kernel and the rendering engine, balancing security, compatibility, and performance. The architecture allocates high-risk components, such as the HTML parser, the JavaScript virtual machine, and the Document Object Model (DOM), to its sandboxed rendering engine. These components are complex and historically have been the source of security vulnerabilities. Running these components in a sandbox helps reduce the severity of unpatched vulnerabilities in their implementation. The browser kernel is responsible for managing persistent resources, such as cookies and the password database, and for interacting with the operating system to receive user input, draw to the screen, and access the network. The architecture is based on two design decisions:

1. The architecture must be compatible with the existing web. Specifically, the security restrictions imposed by the architecture should be transparent to web sites. This design decision greatly limits the landscape of possible architectures but is essential in order for Chromium to be useful as a web browser. For example, the architecture must support uploading files to web sites in order to be compatible with web-based email sites that let users add attachments to emails.
2. The architecture treats the rendering engine as a black box that takes unparsed HTML as input and produces rendered bitmaps as output (see Figure 1). In particular, the architecture relies on the rendering engine alone to implement the same-origin policy. This design decision reduces the complexity of the browser kernel's security monitor because the browser kernel need only enforce coarse-grained security restrictions. For example, the browser kernel grants the ability to upload a

CHROMIUM ARCHITECTURE



Rendering Engine:

Interprets and executes web content

Outputs rendered bitmaps

The website is the “untrusted code”

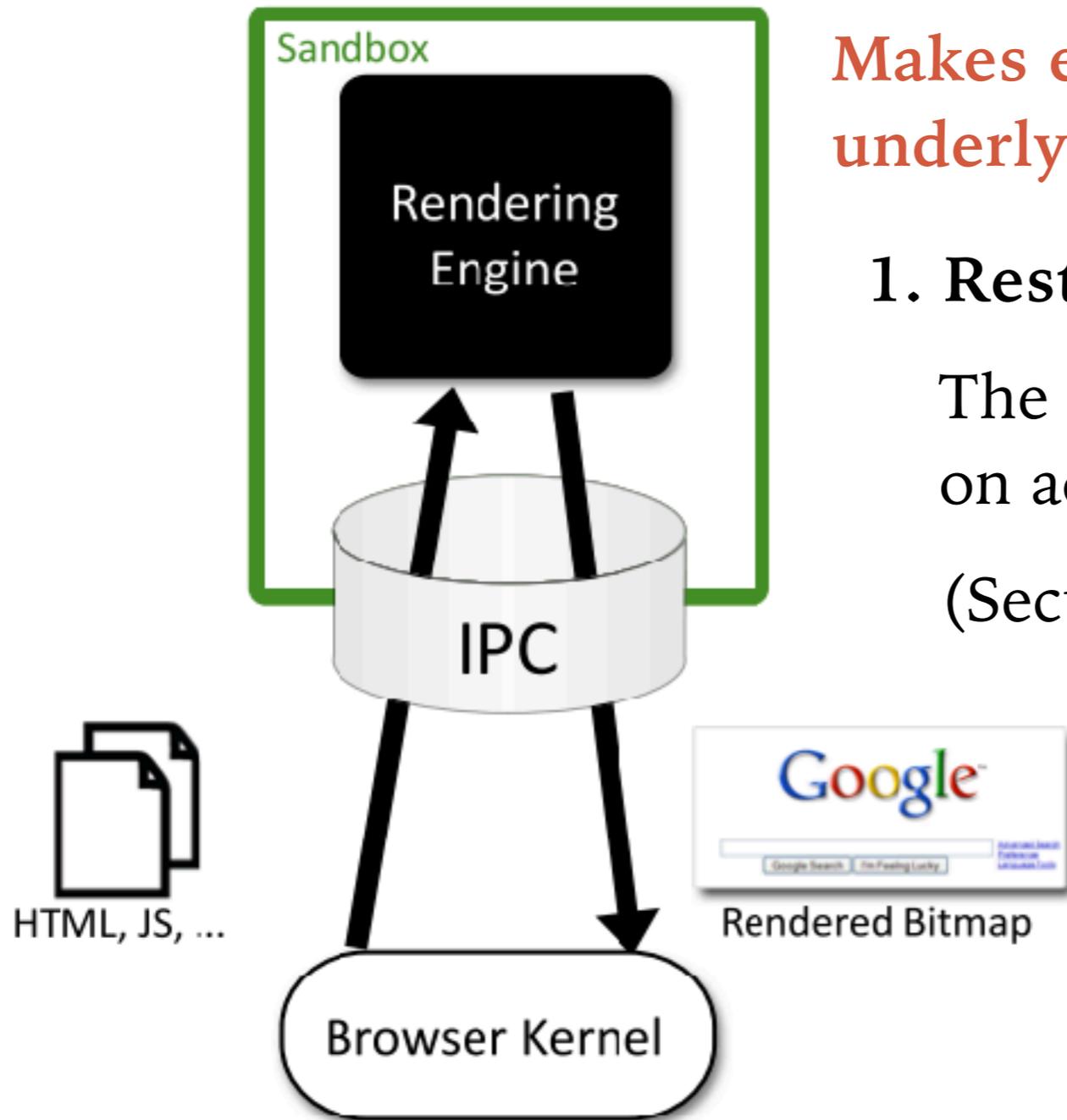
Goal: Enforce a narrow interface between the two

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)

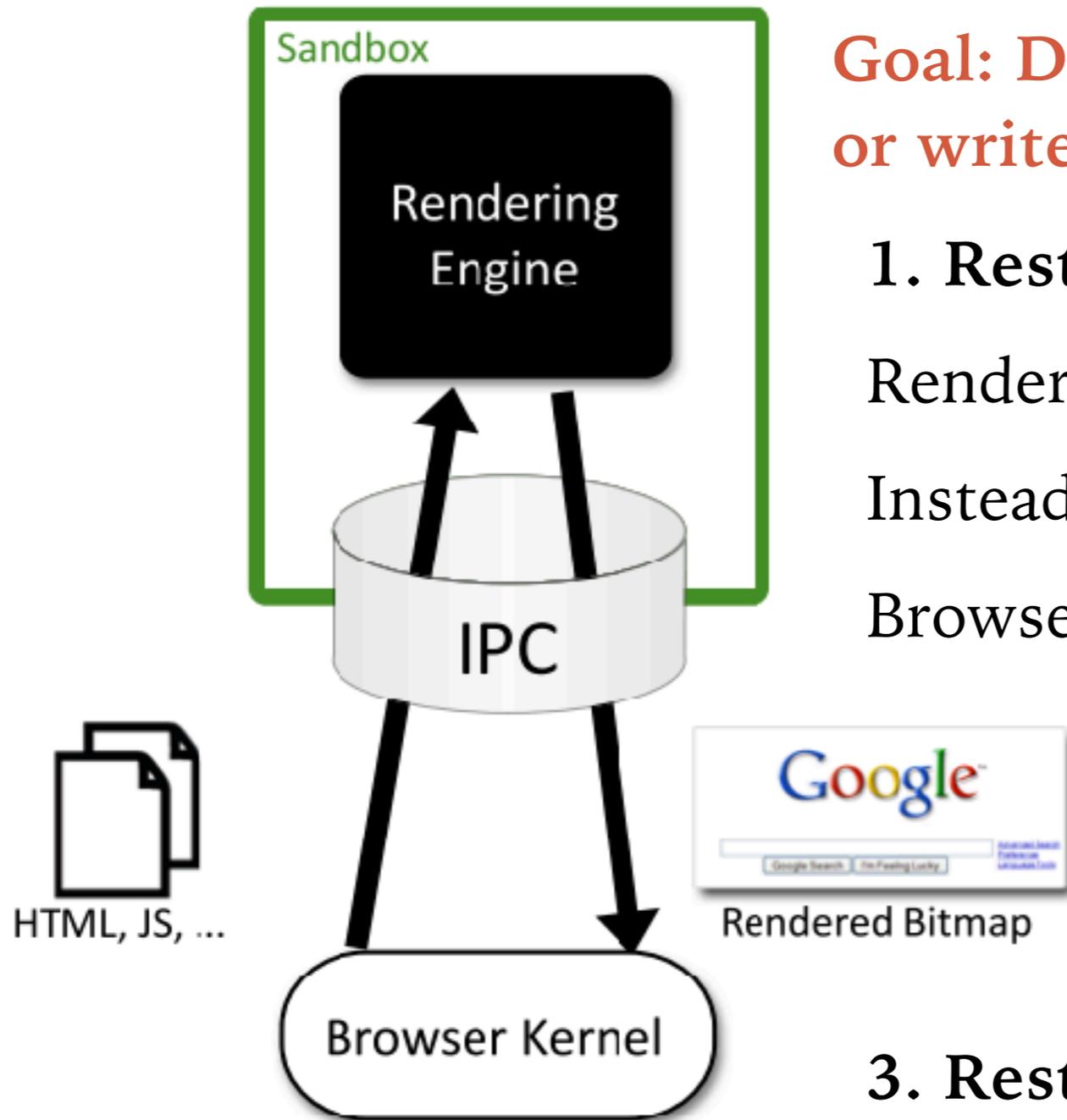
2. Separate desktop

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