Enforcing the Principle of Least Privilege in Web Servers via Software-Based Fault Isolation

David Ansari, Yen-Pin Huang, George Robinson, Tiancheng Yi

Supervisor: Prof. Brad Karp

MSc Networked Computer Systems

University College London

September 1, 2015

This report is submitted in partial fulfillment of the requirement for the MSc Degree in Networked Computer Systems at University College London. It is the result of our own work except where stated otherwise.

This report will be distributed to the internal and external examiners, but thereafter shall not be copied or distributed without the permission of all original authors.

Abstract

Modern web servers must be fast and secure. To limit the extent of damage that may result from the exploitation of software vulnerabilities, state of the art research [Kro04] [BMHK08] splits web servers into reduced-privilege compartments isolated via separate Unix processes.

However, the creation of and communication between processes has a considerable hit on web server performance. Therefore, we take a different approach. To the best of our knowledge, we build the first reduced-privilege web server isolated via Software-Based Fault Isolation (SFI). We sandbox the HTTP parser for the Monkey HTTP server with Google Native Client (NaCl): an implementation of SFI originally designed for the safe execution of untrusted, native code in the browser.

The results from our evaluation show that the isolation of reduced-privilege compartments via SFI provides both stronger security and better performance than isolation via processes. With isolation via SFI, our modified Monkey HTTP server achieves an increase in sustained requests per second of 33% and a decrease in median latency of 60% compared to isolation enforced with processes. We identify numerous optimizations which we conjecture to further increase performance in future work without compromise to the isolation required between reduced-privilege compartments.

Acknowledgments

We thank our supervisor, Prof. Brad Karp, without whom this project would not have been possible. His devotion to network and systems research has instilled inspiration and drive in us as individuals whilst his dedication, enthusiasm and guidance towards our project has been instrumental to its success.

Last, we would like to thank those most close to us for their unquestionable love and support in all our endeavors for which we are forever indebted.

Contents

List of Figures

List of Tables

Listings

Abbreviations

ABI Application Binary Interface

ADT Abstract Data Type

ASLR Address Space Layout Randomization

CGI Common Gateway Interface

CISC Complex Instruction Set Computing

CWE Common Weakness Enumeration

GCC GNU Compiler Collection

GDB GNU Debugger

HTTP Hypertext Transfer Protocol

IMC Inter-Module Communication

IPC Inter-Process Communication

LLVM Low-level Virtual Machine

NaCl Native Client

OKWS OK Web Server

OS Operating System

PNaCl Portable Native Client

RISC Reduced Instruction Set Computing

SFI Software-Based Fault Isolation

SSL Secure Sockets Layer

TCB Trusted Computing Base

TLB Translation Lookaside Buffer

1 | Introduction

The number of software vulnerabilities both discovered and reported has increased dramatically in the last two decades. Whereas just 25 vulnerabilities were reported in 1995, over 5000 vulnerabilities were reported in 2012 [You13, p. 3]. Whilst past research efforts have proposed a number of techniques to address this issue we know of no panacea which exists to-date that prevents all realms of vulnerabilities found in the wild.

The Common Weakness Enumeration (CWE) database [1] lists SQL injection, Operating System (OS) command injection and buffer overflows as the three most prevalent weaknesses found in software. Most susceptible to buffer overflows is software written in memory-unsafe languages such as *C*, in which the Apache HTTP server, Nginx, lighttpd and many other web servers are written.

One of the most frequent sources of buffer overflow vulnerabilities in memory-unsafe languages is string manipulation, for which *C* is most notorious. It is therefore unfortunate that the Hypertext Transfer Protocol (HTTP) protocol, over which the World Wide Web operates, is a text-based protocol, whilst almost all web servers that implement it are written in *C* themselves.

To exacerbate the situation even further, text-based protocols seldom enforce strict message formats such as a minimum length on requests and the positioning and order of header fields [FGM+99, 4.2]. As a result, web servers require extensive use of string manipulation functions in order to parse and dispatch HTTP requests which ultimately leaves them vulnerable to all sorts of unstructured and untrusted user input.

The Apache HTTP server, currently the world's most popular web server[2], contains over 1.5

[1] *Common Weakness Enumeration*. The MITRE Corporation, https://cwe.mitre.org/top25/index.html, last accessed 03/08/2015

[2] *June 2015 Web Server Survey*. Netcraft, http://news.netcraft.com/archives/2015/06/25/june-2015-web-server-survey.html, last accessed 03/08/2015

million lines of code[3], thousands of which process untrusted user input. The presence of a single vulnerability in one line amongst thousands could compromise the confidentiality, integrity and availability of the entire server. Once compromised, there is little to prevent a sufficiently advanced exploit from leaking and modifying sensitive information including private user data and cryptographic keys.

Techniques developed to detect the presence of software vulnerabilities include formal methods such as static analysis. However, despite previous efforts to exercise static analysis in the development of secure web applications [HYH+04], static analysis is not widely used in the identification of vulnerabilities in web server software. One such limitation of formal methods includes the state explosion problem [CKNZ11] in the presence of concurrent threads and processes implemented to build high-performance servers.

Rather than attempt to eliminate vulnerabilities from server-side software entirely we can instead mitigate the degree to which vulnerabilities can be exploited. This idea underpins the Principle of Least Privilege which states that "every program and every user of the system should operate using the least set of privileges necessary to complete the job." [SS75, p. 5].

Previous works [Kro04] [BMHK08] enforce the Principle of Least Privilege for web server software by splitting the server's code into reduced-privilege compartments. One such example of a reduced-privilege compartment is a web server's HTTP parser whose role it is to process untrusted user input. Since an SSL session must be established prior to receiving the HTTP request, there is no need for the HTTP parser to have access to sensitive data such as cryptographic keys. However, as the HTTP parser runs in the same address space as the rest of the server there is little to prevent a successful exploitation of the HTTP parser from stealing or modifying sensitive data. The motivation behind least privilege is therefore to separate the HTTP parser from the rest of the server process. Thus, even if the HTTP parser is vulnerable, a successful exploitation cannot compromise sensitive data or the control flow of the rest of the server.

One natural option is to leverage the existing OS isolation primitives between processes as a means to separate reduced-privilege compartments. This is how compartments are isolated in the aforementioned previous works [Kro04] and [BMHK08]. However, one significant drawback of enforcing least privilege via Unix processes is the large number of context switches for frequently communicating compartments. In fact, a simple message exchange between two processes involves expensive operations such as (re)storing registers, the program counter, stack pointer,

[3] *Apache HTTP Server on OpenHub*. OpenHub, https://www.openhub.net/p/apache, last accessed 03/08/2015

memory maps and flushing the Translation Lookaside Buffer (TLB). As a consequence, performance suffers for frequently communicating reduced-privilege compartments separated by process boundaries. The question therefore arises whether we can isolate reduced-privilege compartments whilst avoiding the performance hit that results from using Unix processes.

One such technique is software-based fault isolation (SFI). SFI enables the safe execution of untrusted code (including a potentially faulty HTTP parser) within the same address space as trusted code and other reduced-privilege compartments through a set of special instrumentation instructions intertwined in the untrusted code. This instrumentation of untrusted code is colloquially known as *sandboxing*. Instrumented untrusted code is therefore said to be *sandboxed*.

We define *trusted code* as being part of the Trusted Computing Base (TCB) and therefore rely on this code being free from vulnerabilities. Provided that the TCB is kept small, code in the TCB can be rid of vulnerabilities through a combination of static analysis, extensive auditing and thorough code review. In contrast, *untrusted code* does not belong in the TCB and therefore may contain one or more vulnerabilities.

1.1 Sandboxing the HTTP parser and SSL session setup

Two of the most vulnerable modules of an HTTP server are its HTTP parser and Secure Sockets Layer (SSL) session setup both of which process untrusted user input. If one of these modules is exploited, it can be used to leak confidential information such as session cookies and the private RSA exponent for SSL, or to eavesdrop and manipulate both unencrypted and secure client sessions, such as modifying GET requests, POST data, and HTTP responses.

The HTTP parser is tasked with processing untrusted user input in the format of an HTTP request. Since the HTTP protocol is text-based the parsing of an HTTP request often involves significant usage of string manipulation functions. However, for performance, most HTTP servers are written in memory-unsafe languages such as C and C++. We show in Section 2.3.3 how string manipulation functions can be abused to exploit vulnerabilities. In practice, there is no reason for granting the HTTP parser permission to read and write from network sockets, the file system or interact with components of the HTTP server other than its caller. Nevertheless, most HTTP servers in use today do not enforce the principle of least privilege giving the HTTP parser the privileges to do that.

The SSL setup component is tasked with the setup of an SSL connection between client and server. Recall from the SSH handshake [FKK11, 5.6] that the client initiates the session with a `Client Hello` message. The server responds with a `Server Hello` message, followed with a `Server Certificate` and `Server Hello Done` message. The client then encrypts a pre-master secret using the server's public key sending it to the server. Next, both the client and the server use the client random, server random and pre-master secret in order to generate a master secret: a shared secret between the two parties used for symmetric encryption and message authentication codes. Finally, client and server exchange their respective `Change Cipher Spec` and `Finish` messages.

Should an attacker be able to exploit the SSL setup component of an HTTP server, then the server could be compromised in multiple ways. First, the attacker could leak the SSL private key from the server process' address space to a file descriptor of a network connection associated with the adversary.

In another example the attacker could read both the pre-master secret and the master secret from the HTTP server's internal data structures for a given SSL session enabling the attacker to eavesdrop or execute a man-in-the-middle attack on the victim.

In a final example, the attacker could overwrite the value of the server random sent to the client in the `Server Hello` message. Since the attacker can determine the value of the server random he can force the server to use the server random of a past session and hence replay any past SSL session to the server.

We now describe the problem this report seeks to address.

1.2 Problem description

Prior implementations of reduced-privilege web servers enforce the Principle of Least Privilege through the splitting of server code into separate reduced-privilege Unix processes (and variants of). However, due to the nature of processes themselves these implementations suffer from expensive inter-compartment communication and the cost of process creation.

We present the following problem statements:

1. Can we therefore increase the performance of a web server that separates reduced-privilege compartments via SFI rather than Unix processes?

It is our hypothesis that enforcing the Principle of Least Privilege via SFI yields better performance than equivalent fault isolation via Unix processes.

2. What is the cost of sandboxing reads in addition to sandboxing writes, and is that cost acceptable for high-performance web servers?

If the cost of sandboxing reads is unacceptable we might want to forgo the guarantee of confidentiality in exchange for an increase in performance.

1.3 Threat model

We now introduce the threat model upon which our work is based. We state the assumptions we make of our adversary, trusted and untrusted code.

1. All code within the TCB is free from vulnerabilities and cannot be exploited.
2. All non-reduced-privilege server code is part of the TCB.
3. The source code of reduced-privilege compartments may contain vulnerabilities.
4. The object code of a reduced-privilege compartment has the correct instrumentation (sandboxing) instructions as a result of the SFI compilation.
5. An adversary can send arbitrary data to the web server.
6. An adversary can exploit all vulnerabilities in a reduced-privilege compartment. He can read and write all data belonging to the reduced-privilege compartment's data, heap and stack, and can jump to all instructions in the reduced-privilege compartment's code.

1.4 Goals

The overall objective of our project is to demonstrate that we can use SFI to build a reduced-privilege web server with better performance than the same server isolated via separate Unix processes. To the best of our knowledge, no HTTP server exists to-date which enforces the Principle of Least Privilege via SFI.

In order to help succeed in our project and distribute work amongst our group we divide our objective into four high-level goals. We classify each goal with one of two priorities. Goals

classified as *critical goals* are critical to our project and must be achieved within the three month project time frame. Goals classified as *non-critical goals* are of lesser importance. However, we hope to achieve these goals given sufficient time remaining having satisfied all critical goals first.

We then subdivide each of our four goals into three manageable parts: *purpose*, *advantage* and *measurement*. The *purpose* outlines what we intend to achieve. The *advantage* lists the benefits of achieving it whilst *measurement* states how we intend to quantify the *advantage*.

1.4.1 Critical goals

Goal 1

- **Purpose:** To separate out the HTTP parser of a simple, 100 lines of code web server into a reduced-privilege compartment isolated via SFI.
- **Advantage:** An attacker should be unable to compromise sensitive information even in the presence of exploitable software vulnerabilities within a reduced-privilege compartment. SFI should yield lower communication costs between trusted code (the rest of the simple web server) and untrusted code (the HTTP parser) when compared to fault isolation via Unix process. As a result, the reduced-privilege SFI simple server should be faster than the same reduced-privileged simple server isolated via Unix processes.
- **Measurement:** To demonstrate that an attacker cannot compromise sensitive information in the presence of vulnerabilities we first develop a working buffer overflow exploit for the non-sandboxed HTTP parser. When exploited, the HTTP parser transfers control to a privileged function outside the normal control flow of the server. We port the same exploit to the reduced-privilege, SFI sandboxed HTTP parser (with all privileged functions in trusted code) and demonstrate that an exploit of the parser cannot transfer control to the privileged functions in the trusted code. To measure the communication cost between reduced-privilege compartments for SFI and Unix processes we set up microbenchmarks in which we measure and compare the execution times for a series of message exchanges. To measure whether the reduced-privilege SFI simple server is faster than the same server isolated via Unix processes we run a series of macrobenchmarks in which we measure the throughput [requests per second] sustained from each server as the offered load increases.

Goal 2

- **Purpose:** To sandbox the HTTP parser of an HTTP 1.0/1.1 compliant, feature-complete[4], concurrent web server.
- **Advantage:** To demonstrate that SFI is a suitable mechanism for enforcing least privilege in large and complex software systems used in real world deployments.
- **Measurement:** Like with Goal 1, we measure whether the reduced-privilege SFI server is faster than the same server isolated via Unix processes through a series of macrobenchmarks.

Goal 3

- **Purpose:** To measure the overhead (if at all) of sandboxing reads in addition to sanboxing writes for untrusted code.
- **Advantage:** To understand whether sandboxing reads in addition to writes reduces the overall performance of sandboxed code and whether the reduction in performance (if at all) is acceptable for high-performance HTTP servers.
- **Measurement:** Measure the maximum attainable throughput [requests per second] and compare the total number of instructions executed between an HTTP server whose reduced-privilege compartments are compiled with only sandboxed writes vs. the same HTTP server whose reduced-privilege compartments are compiled with sandboxed reads and writes.

1.4.2 Non-critical goals

Goal 4

- **Purpose:** Sandbox the SSL module of the HTTP server from Goal 2.
- **Advantage:** Prevent the compromise of confidential information for an SSL session or the SSL private key.
- **Measurement:** An attacker should not be able to compromise confidential information from an SSL session other than its own nor compromise long-term confidential information

[4]We define a feature-complete web server as a web server that has the minimum set of features expected of a fit for purpose web server.

such as the SSL private key.

1.5 Structure of this report

We now outline the structure for the remainder of this report. Chapter 2 establishes background knowledge that is essential in order to understand our work. In Chapter 3, we discuss the design and implementation of our efforts towards SFI for reduced-privilege web servers. Chapter 4 presents an evaluation of our work, including both micro and macro benchmarks to compare the performance between reduced-privilege web servers isolated via SFI and the equivalent server isolated via Unix processes. In Chapter 5, we summarize our contributions and propose future work in Chapter 6. Chapter 7 outlines our project management process including our schedule, team organization, communication and risk management. We conclude in Chapter 8.

2 | Background

This chapter starts with a review of the memory layout of C programs which is fundamental to the successful exploitation of vulnerable software. In order to leverage this knowledge in the development of exploits we explain the fundamentals behind buffers in C and demonstrate how an attacker can overrun a buffer to exploit running software, such as a web server.

Next, we introduce the Principle of Least Privilege as an effective primitive to constrain exploits via the separation of software into least privilege compartments. We discuss two previous works that enforce the Principle of Least Privilege using Unix processes (and variants thereof) which leads onto our introduction of SFI: a fault isolation primitive for running least privilege compartments within the same address space (and hence the same Unix process). Last, but not least, we introduce NaCl: an implementation of SFI to enable the secure execution of untrusted, x86 and x86-64 native code in the browser.

2.1 The memory layout of C programs

All processes in Unix have their own private address space. Therefore, from the perspective of a process, each process owns all addresses in the address space ($[0, 2^{32})$ on 32-bit systems and $[0, 2^{48})$ on 64-bit systems[5]). However, this address space is virtual. The OS just provides the illusion that every process owns all addresses in the address space. This illusion is better known as virtual memory.

To support the requirements of virtual memory the kernel manages a page table for every process known to the OS. This page table maps the virtual addresses in the process' address space to addresses in physical memory. Since no two processes can use the same physical addresses to

[5] *AMD64 Architecture Programmer's Manual*. AMD, http://developer.amd.com/wordpress/media/2012/10/24593_APM_v2.pdf, last accessed 28/08/2015

store different data, the same virtual addresses for a given pair of processes often map to different physical addresses; shared libraries being the exception. However, since the virtual address space is so large, the kernel must swap pages to disk when it finds that the amount of available physical memory is less than the amount of virtual memory demanded from its processes.

A process divides its virtual address space up into regions, known as segments. At the bottom of a process' address space is the code segment, otherwise referred to as the text segment. The instructions which tell the CPU how to run the program are loaded here and are known as the program text.

Directly above the code segment is the data segment. The data segment contains all initialized global and static variables. All uninitialized global and static variables are stored in the BSS segment. All pages in the BSS segment are initialized to zero and hence all uninitialized global and static variables are initialized to zero too. The BSS segment tails the data segment.

The heap starts immediately above the BSS segment. Unlike the program text, global and static variables, which are declared at compile time, the heap is unfixed. It can increase and decrease in size, growing towards the higher addresses in the virtual address space. Unlike the code, data, BSS and stack, the heap is unmanaged. In effect, it is a "free-floating"[6] region of the address space where dynamic storage is allocated. Unlike the stack, where the allocation and de-allocation of variables is automatic, storage allocated on the heap must be returned via a call to free() once it is no longer needed. Heap allocated storage which is not returned to the heap is known as a *memory leak* and can result in programs accumulating excessive amounts of memory over time.

Last but not least is the stack. The stack starts at the top of the address space where it grows down towards the lower addresses [TB15, p. 757]. On some platforms, it is possible for the stack and the heap to collide. Upon collision the program will either crash in the presence of guard pages or silently corrupt existing data[7].

The stack itself is a last-in first-out (LIFO) Abstract Data Type (ADT). There exist just two operations that can be performed on a stack: PUSH and POP. The PUSH operation places the element on the top of the stack, whilst the POP operation removes and returns the element from the top of the stack.

[6] *Stack v.s. Heap*. Paul Gribble, http://gribblelab.org/CBootcamp/7_Memory_Stack_vs_Heap.html, last accessed 12/08/2015

[7] *Silent stack-heap collision under GNU/Linux*. Vincent Lefevre, https://gcc.gnu.org/ml/gcc-help/2014-07/msg00076.html, last accessed 12/08/2015

The stack is a fundamental building block of high-level programming languages. High-level languages, such as C and C++, support a construct known as the function: a sequence of basic blocks whose scope is separate from all other functions in the program [One96]. The use of functions enables the division of otherwise complex software into logical, well-defined, structured units of execution.

The invocation of a function changes a program's control flow and thus modifies the value of its instruction pointer (a special CPU register which holds the address of the next instruction to execute). Since the invocation of a function can cause the instruction pointer to execute at some other address in the program text, known as a jump, the program must track the address of the instruction that should be executed post return from the function: that is, the instruction following the jump. This address is known as the return address.

When a function is invoked the CPU pushes the return address and base pointer (which denotes the start of the stack frame) onto the stack. It then copies the current stack pointer (which denotes the top of the stack) into the base pointer in a procedure known as the function prologue. When the function returns, such that there are no more instructions to execute as part of the function, the program reverses these operations in a procedure known as the function epilogue. Upon completion of the function epilogue the CPU can continue execution exactly where it left off.

However, in addition to return addresses and frame pointers, the stack is also used to store local variables: variables which are neither static, global, nor allocated from the heap. Since the contents of a variable must be modifiable (otherwise it would be a constant) the stack must be both readable and writable. As a result, it is possible for a program to overwrite other variables on the stack including, but not limited to, the saved base pointer and the return address and thus enables a special class of exploits including buffer overflow attacks.

2.2 The Unix process model

In Unix the creation of a process is restricted to a single system call known as fork [TB15, p. 733]. The fork system call duplicates the calling process, including the calling process' code, variables, state, and file descriptors: distinct intra-process integer numbers, each corresponding to an open file in the virtual file system. When the caller, known as the parent, resumes execution from the fork system call, it receives the process ID of the created process, known as the child, as the return value of fork.

Despite the child process being a duplicate of its parent, both the parent and child are seen as two distinct processes. A state transition in the parent process does not affect the child process, nor vice-versa. In order to enforce this requirement the kernel must copy the code, data, stack and heap from the parent process to the child process. However, copying is expensive and has a time and space complexity proportional to the amount of data to be copied. To avoid this cost, the kernel creates a separate page table for the child process and points each of its entries to the pages of the parent process. The kernel then marks each of the pages as read-only, such that the first process to write to a given page will cause a protection fault. On fault, the process traps to the kernel which only then creates a copy of the page for the instigating process, marking the original page as read-write for the other [TB15, p. 742]. This is known as *copy-on-write*.

However, whilst fork is a convenient and fast method to create a process, quite often the caller will want to start a program other than its own. In order to fulfill this need, Unix supports a system call which replaces a process' core image with that of another executable. This system call, known as exec, takes the name of an executable file and copies the arguments and environment variables to the kernel. The kernel releases the process' old page table and creates a new unmapped page table backed by the executable file. When the process starts to run it will initiate a page fault, causing the kernel to map the first page of the executable file into the process' address space. The process will continue to page fault as it attempts to execute, read or write from unmapped pages until all unmapped pages have been mapped in [TB15, p. 742].

2.3 Buffers and exploits

A buffer is a contiguous sequence of addresses in the virtual address space. It is an array storing ephemeral data such as a sequence of bytes and hence is declared in C as of type char array. Whilst a buffer might appear to be identical to a string, a buffer is not required to be of type char array. Furthermore, a string tends to store alpha-numeric characters terminated with a NULL terminator (0x00) whereas no such assumption can be made about a buffer.

When a program reads or writes data from a file, standard input/output (I/O), network socket, or other source of I/O, it will often read or write the data in chunks. This is known as buffered I/O. When a program which uses buffered I/O calls read or recv on a source of input it must include the address of the buffer and its size in bytes. Without this information the read and recv functions do not know where it should write the chunk, nor the maximum size of each chunk it

can write. Both the read and recv functions return the number of bytes copied into the buffer since the number of bytes can be less than the maximum size of a chunk. Upon returning from read or recv, the program is free to process the chunk as it sees fit. It can then call read or recv to have the next chunk copied into the buffer, continuing in this manner until there is no more data remaining as indicated via the EOF (end of file) character.

2.3.1 Overflowing a buffer in C

A buffer overflow occurs when a program writes data beyond the end of a buffer. For example, for a buffer of 255 bytes, a buffer overflow would occur should the program write 256 bytes into the buffer. The reason buffer overflow attacks are so dangerous is because most buffers are allocated on the stack. Since data is written from low address to high address it becomes possible to write past the end of the buffer and overwrite the return address of the current stack frame, thus changing the program's control flow.

Whilst it is well known that the string and memory manipulation functions in C enable programs to overwrite past the end of the buffer, some functions are safer than others. However, if used incorrectly, the safer manipulation functions can still result in buffer overflows. For example, whilst the memset(), memcpy(), read(), recv() and strncpy() functions require the programmer to pass the length of the buffer as an argument, it is possible for these functions to write past the end of the buffer should the programmer pass an incorrect length or pass the correct length but a location other than the start of the buffer. Other functions, such as strcpy(), are inherently unsafe. The strcpy() function will continue to copy bytes from the source into the destination buffer until a special character, known as a NULL terminator, is encountered in the source. Therefore, if the source is much longer than the buffer it will write past the end of the buffer until all bytes from the source have been copied.

2.3.2 Mitigations

At the time when buffer overflows were still in their infancy [One96] it was possible to read, write and execute data on the stack. Hence, some of the most effective buffer overflow exploits injected code into running programs. With the instructions of the injected code stored in the buffer, the exploit would then continue to overflow the buffer and overwrite the return address to point to the start of the injected code. Then, when the function returned it would start to execute the

injected code rather than the instructions first intended.

However, CPUs have since added support for a feature known as executable space protection, also known as *Write XOR Execute*. This feature enables the OS to set pages as executable or writable, but not both. Hence, whilst an attacker can still inject code into the stack, their code cannot be executed since all virtual pages for the stack are marked as read-write, no execute; thus causing the program to crash. Whilst this ensures an attacker cannot execute injected code in a running program, it still enables an attacker to perform denial-of-service attacks against vulnerable software.

In order to make the exploitation of such vulnerabilities even more difficult, OSs introduced a feature known as Address Space Layout Randomization (ASLR). ASLR randomizes the base address of a program at load time such that the addresses of stack variables (and program text if compiled as a relocatable program) differ by some random base address between executions.

In fact, when employed with write-no-execute and stack canaries, ASLR proves to be a formidable outer-layer of defense. Furthermore, it requires no modifications to existing software.

However, these primitives are no panacea. It has been shown that the randomization is insufficient for 32-bit software [SPP+04]. Furthermore, a number of software systems make it easier for attackers to predict the randomization offset [SPP+04]. In particular, HTTP servers that use the worker and pre-fork execution models instantiate their child processes via a call to fork and thus preserve the same randomized base offset across their children. This lack of re-randomization provides attackers with an infinite time frame in which to predict the random base offset. Should a child process crash, as often results from an attempted exploit, the parent will detect the child has crashed and re-fork the child with the same random offset, thus enabling the attacker to try different random offsets indefinitely.

2.3.3 Exploiting buffer overflows

Before we introduce the Principle of Least Privilege we first demonstrate how to exploit a buffer overflow vulnerability in order to change the values of other variables on the stack. In particular, we show that we can change the value of an integer variable *modified* from 0 to 1 by writing past the end of a 64 byte buffer that reads characters from standard input via the unsafe libc gets

function.

```
int main() {
    int modified;
    char buf[64];
    modified = 0;
    gets(buf);
    printf("The value of modified is: %d\n", modified);
    return 0;
}
```

Listing 2.1: C program with a buffer overflow vulnerability.

Listing 2.1 shows the vulnerable program that we intend to exploit via buffer overflow. It first declares an integer variable *modified* and a 64 byte buffer *buf* on the stack. It then initializes the value of *modified* to 0 and reads characters from standard input into *buf* via the libc gets() function. Finally, it prints the value of the variable *modified* and returns with an exit status of 0.

We compile the program from Listing 2.1 with the GNU Compiler Collection (GCC) compiler on Linux x86-64 and set the -fno-stack-protector flag in order to disable stack protection. Whilst this particular buffer overflow exploit works with ASLR enabled, we disable ASLR for all subsequent exploits where we overwrite the return address stored on the stack. However, we reiterate that stack protection, Write XOR Execute and ASLR are not guaranteed to prevent the exploitation of software. Instead, their collective purpose is to render exploit development impractical. We therefore disable these protection mechanisms in order to ease the exploitation of running software for the purpose of demonstration.

With the program from Listing 2.1 compiled we can now start to develop our exploit. However, before we can overflow *buf* and change the value of the variable *modified* from 0 to 1 we must first find the location of *buf* and *modified* on the stack. In order to find the offset from the start of *buf* to the start of *modified* we examine the object code of the compiled x86-64 binary with

the objdump tool (see Listing 2.2).

```
000000000040057d <main>:
  40057d: 55  push %rbp
  40057e: 48 89 e5  mov %rsp,%rbp
  400581: 48 83 ec 50  sub $0x50,%rsp
  400585: c7 45 fc 00 00 00 00  movl $0x0,-0x4(%rbp)
  40058c: 48 8d 45 b0  lea -0x50(%rbp),%rax
  400590: 48 89 c7  mov %rax,%rdi
  400593: e8 e8 fe ff ff  callq 400480 <gets@plt>
  400598: 8b 45 fc  mov -0x4(%rbp),%eax
  40059b: 89 c6  mov %eax,%esi
  40059d: bf 44 06 40 00  mov $0x400644,%edi
  4005a2: b8 00 00 00 00  mov $0x0,%eax
  4005a7: e8 a4 fe ff ff  callq 400450 <printf@plt>
  4005ac: b8 00 00 00 00  mov $0x0,%eax
  4005b1: c9  leaveq
  4005b2: c3  retq
  4005b3: 66 2e 0f 1f 84 00 00  nopw %cs:0x0(%rax,%rax,1)
  4005ba: 00 00 00
  4005bd: 0f 1f 00  nopl (%rax)
```

Listing 2.2: Object dump of the code shown in Listing 2.1.

Listing 2.2 shows an object dump of the compiled x86-64 binary. The instruction at address 400581 shows that 80 bytes (0x50) are reserved on the stack for *buf* and *modified*. Given that *buf* is 64 bytes and *sizeof* int is 4 bytes there must be 12 bytes of padding between the end of *buf* and the start of *modified*: consistent with the 12 byte stack alignment on x86-64 and confirmed by the instruction at address 400598 which loads *modified* into the register %eax. Hence, we must write 76 bytes into *buf* in order to reach the start of *modified*. We can then add a 77th byte to our exploit in order to overwrite the least significant byte of *modified* and change its value. This overwrites the least significant byte rather than the most significant byte because both x86

and x86-64 are little endian and therefore store the least significant byte first.

```
echo -en "$(perl -e "print 'A'x76")\x01" | ./main
```

Listing 2.3: Complete exploit string to overflow *buf* and change the value of *modified* from 0 to 1.

Listing 2.3 shows our finished exploit. It first writes the character 'A' 76 times. This run of 'A's consumes the 64 bytes in *buf* and the 12 bytes of padding that follow. It then writes a 1 (0x01) for the 77th byte in order to change the least significant byte of *modified* from 0 to 1 as demonstrated in the output shown in Listing 2.4.

```
echo -en "$(perl -e "print 'A'x76")\x01" | ./main
The value of modified is: 1
```

Listing 2.4: This trace shows the output of Listing 2.1 when exploited to overwrite the least significant byte of *modified*.

However, sophisticated exploits often need to overwrite more than just the least significant byte. Instead, such exploits must overwrite the entire contents of a variable or address stored on the stack. Since x86 and x86-64 are little endian we can overwrite the entire contents of a variable or address by overwriting the values in reverse, least-significant byte first. For example, the exploit in 2.5 changes the value of *modified* from 0 to 256. It uses the 77th byte to overwrite the least significant byte of *modified* with a 0 and the 78th byte to overwrite the second least significant byte with a 1, thus changing the value of *modified* to 256.

```
echo -en "$(perl -e "print 'A'x76")\x00\x01" | ./main
The value of modified is: 256
```

Listing 2.5: This trace shows the output of Listing 2.1 when exploited to overwrite the two least significant bytes of *modified*.

Despite providing a formidable outer-layer of defense, the mitigations discussed in Section 2.3.2 are not guaranteed to prevent the exploitation of software. Instead, their collective purpose is to render exploit development impractical. Thus, we cannot assume that software is unexploitable in the presence of these mechanisms. Furthermore, previous works [SPP+04] have shown that primitives such as ASLR are ineffective on 32-bit address spaces and software that uses the fork-join model such as the Apache HTTP server.

Therefore, a more realistic outlook is to accept that software *will* be exploited and that once exploited damage must be limited to the exploited compartment. This sentiment underpins the Principle of Least Privilege which we will now discuss.

2.4 The Principle of Least Privilege

The Principle of Least Privilege is the principle that all components within a software system run with the minimal privileges necessary to fulfil their purpose [SS75, p. 5]. For example, an HTTP server that enforces the Principle of Least Privilege might split its functional components into least privilege compartments each tasked with a distinct role in the request-response cycle of an HTTP server.

Hence, in order to enforce the Principle of Least Privilege each of an HTTP server's compartments must run with the least privileges possible. Each compartment must be restricted such that it cannot read from or write to other least privilege compartments except via secure inter-compartment communication implemented by a well-defined interface. Furthermore, each compartment must be restricted such that it cannot jump and execute the code of least privilege compartments other than itself.

However, since each least privilege compartment fulfils a distinct role in the request-response cycle of an HTTP server, we require that elevated permissions are granted on the granularity of compartments. For example, a least privilege SSL connection setup compartment will need permissions to read the SSL private key exponent from the file system yet it must be denied permissions to read or write from a network socket as this would enable the compartment to leak sensitive information such as the private key. Likewise, a least privilege HTTP request parser compartment must neither be able to read or write from a network socket nor interact with the file system.

This compartmentalization of functional components underpins the Principle of Least Privilege and is fundamental to its effectiveness in enforcing fault isolation.

Perhaps the most intuitive option is to leverage existing OS primitives and isolate compartments as separate Unix processes. We discuss two previous works that adopt this option and then introduce SFI which enables fault isolation within a single Unix process.

2.5 Previous works

In this section we introduce two previous works that use the Principle of Least Privilege to build secure HTTP servers. The first, the OK Web Server (OKWS) splits compartments into least privilege Unix processes. The second, Wedge, enforces least privilege via a variant of Unix processes known as *sthreads*.

2.5.1 OK Web Server

The OK Web Server (OKWS), is a secure, high performance web server optimized for dynamic content, such as personalized HTML pages [Kro04, p. 1]. The OKWS provides a set of shared libraries and helper processes to receive and dispatch HTTP requests, access data sources such as SQL databases, log events and generate HTML-formatted responses [Kro04, p. 3].

In order to enforce the Principle of Least Privilege each compartment (whether a helper process or application specific executable), referred to as a service, runs in a separate chrooted, unprivileged Unix process [Kro04, p. 4]. The OK launcher daemon process is responsible for starting and supervising all least privilege compartments. To start a least privilege compartment the launcher daemon invokes the fork system call. In the child it sets the appropriate `uid` and `gid` pair before `chroot()`ing to a distinct remote jail. The child process then calls exec to run the executable for the appropriate compartment.

However, one limitation of OKWS is that it creates one least privilege process per compartment shared across multiple users (which we refer to soft isolation in Section 3.1) rather than one least privilege process per compartment per user (which we refer to as strict isolation in Section 3.1). The difference is subtle yet important. The first provides isolation between least privilege compartments such that the exploitation of one least privilege compartment cannot coerce other least privilege compartments. However, the second provides isolation between users of least privilege compartments should a compartment become compromised.

The OKWS chooses the former isolation guarantee (i.e. soft isolation) in order to demonstrate reasonable performance. Therefore, an attacker who exploits a least privilege compartment could still learn and influence the data of existing or subsequent sessions belonging to other users.

2.5.2 Wedge

Wedge is a system to aid the splitting of complex software, such as the Apache HTTP server and OpenSSL, into least privilege compartments. At its core, Wedge enforces the Principle of Least Privilege using a *default-deny* model [BMHK08, p. 1] which requires that privileges are made explicit for each least privilege compartment.

Wedge itself provides a number of OS level primitives to create and manage least privilege compartments, otherwise known as *sthreads*. On creation each sthread starts with a clean copy-on-write snapshot of its parent sthread taken prior to the start of the parent's execution. However, unlike with the fork system call, a child sthread inherits zero privileges from its parent. Instead, the parent sthread can grant each of its children with equal or lesser privileges than its own.

2.6 Software-Based Fault Isolation

Compartmentalization via Unix processes (or variants of) is one way to enforce the Principle of Least Privilege. However, whilst effective, communication between processes incurs the overhead of a context switch, copying buffers between kernel and user-space, a mode switch, cache misses and TLB flushes.

SFI enables inter-compartment isolation within the same address space. It therefore avoids the need to isolate compartments via Unix processes and thus eliminates the additional overhead for Inter-Process Communication (IPC).

In SFI, compartments, which are referred to as distrusted modules, are split into isolated fault domains, each of which constituting a distinct subset of the virtual address space.

In order to restrict a fault domain such that it cannot execute code stored in the stack or heap (which would enable a distributed module to bypass fault isolation), the address space of each fault domain is split into two regions: one for the fault domain's code, known as its *code segment*, and the other for the fault domain's stack, heap, static and global variables, known as its *data segment*. The addresses that lie within a given segment all share the same pattern of upper bits, known as the segment identifier, through a set of special alignment rules [WLAG93, p. 3].

Prior to execution of a distrusted module, a special program known as the verifier asserts that the object code within a distrusted module is safe. The verifier is trusted and must be proven

correct. All other code including the SFI compiler can be untrusted which keeps the TCB small.

To ensure that a distrusted module cannot read, write or execute outside its fault domain the verifier statically verifies that all direct jumps, loads and stores refer to the correct segment within the distrusted module's fault domain.

In order to enforce fault isolation for indirect jumps, the compiler inserts a sequence of safe instructions in a procedure known as segment-matching [WLAG93, p. 3]. These safe instructions check whether the segment identifiers for all indirect jumps match the segment identifiers of the distrusted module's code segment and that the segment identifiers for all indirect load and stores match the segment identifiers of the distrusted module's data segment.

Prior to the execution of a distrusted module the verifier verifies that these safe instructions are present for all indirect jump, load and store instructions. A distrusted module cannot manipulate the masking instructions since segment matching uses dedicated registers inaccessible to the distrusted code. Hence, whilst a distrusted module can skip one or more masking instructions it cannot write to the dedicated register which is guaranteed to hold the correct segment identifier. Thus, safety is never compromised.

One drawback of segment matching is that it requires four additional CPU instructions [WLAG93, Figure 1] per indirect jump, load or store. However, the number of additional CPU instructions can be reduced with a method known as address sandboxing. Unlike segment matching, which checks whether the address matches the correct segment identifer, address sandboxing forces the address to have the correct segment identifier. As a result, address sandboxing requires just two additional CPU instructions but at the cost of a fifth dedicated register and decreased debugging information since it cannot identify the faulting instruction.

```
dedicated-reg <= target-reg & mask-reg
dedicated-reg <= dedicated-reg | segment-reg
store instruction uses dedicated-reg
```

Listing 2.6: Address sandboxing instructions for indirect jumps, loads and stores [WLAG93].

Listing 2.6 shows pseudocode for the address sandboxing instructions [WLAG93, Figure 2]. The first line performs a bitwise `AND` between the target address and the masking register in order to remove the existing segment identifier and stores the result in the dedicated register. The second instruction performs a bitwise `OR` between the result stored in the dedicated register and

the segment register. This bitwise `OR` adds the correct segment identifier back into the target address and is how address sandboxing can force an address to have the correct segment identifier. The last instruction then completes the intended operation using the forced address stored in the dedicated register.

One limitation of the implementation of SFI described above is that it was developed for the RISC load/store architecture. However, we want to use SFI on x86 and x86-64 CPUs which use the CISC architecture. However, implementing SFI for x86 and x86-64 presents a number of additional problems.

First, both the x86 and x86-64 instruction sets support variable-length instructions. Whilst this enables compilers to reduce the size of compiled code through dense packing of CPU instructions, it also has the side-effect of enabling the instruction pointer to jump into the middle of instructions which would enable a program to bypass fault isolation.

Second, x86-64 CPUs removed support for segments which enable division of the address space into regions through which the original implementation of SFI isolated fault domains.

Rather than port SFI to x86 and x86-64 (a task that would be near impossible to complete within the time constraints of our project), we turned to NaCl which gets introduced in the next section.

2.7 Google Native Client

Google Native Client (NaCl) is an open source implementation of SFI to enable the safe execution of untrusted native code in the browser [YSD+09]. It allows for browser based software, otherwise written in interpreted languages such as JavaScript, to be implemented in compiled languages such as C and C++ and execute at speeds comparable with that of trusted native code. NaCl requires a modified compiler toolchain (GCC or Low-level Virtual Machine (LLVM)) for offline compilation and a program validator for online validation of untrusted code in order to assert whether it is safe for in-browser execution.

2.7.1 SFI for x86

In order to restrict the execution of untrusted native code on x86 and x86-64 the NaCl developers had to enforce a number of additional constraints.

First, unlike MIPS (the CPU architecture supported in the original SFI implementation [WLAG93]) which uses the Reduced Instruction Set Computing (RISC) instruction set in which instructions are all of a fixed length, both x86 and x86-64 use the Complex Instruction Set Computing (CISC) instruction set in which instructions can be of lengths up to 15 bytes. Hence, most of the longer x86 and x86-64 instructions contain shorter valid x86 and x86-64 instructions within their bytecode.

```
25 CD 80 00 00
```

Listing 2.7: Instruction for AND eax 0x80CD in x86.

Listing 2.7 shows an example of such an instruction. Should the program jump to the second byte of the instruction `CD 80` it will invoke a system call and trap to the kernel[8] rather than execute the intended bitwise AND instruction between `eax` and `0x80CD`.

Therefore, in order to prevent untrusted code from bypassing fault isolation through jumping into the middle of valid x86 and x86-64 instructions, the developers of NaCl proposed a number of constraints that must be enforced in order to restrict the jump behavior of untrusted code [YSD+09, p. 4].

The first of which requires that all sequences of instructions in the untrusted code are split into 32 byte aligned bundles. The validator ensures that all direct jumps target the start of a 32 byte bundle in order to prevent untrusted code from jumping into the middle of a valid instruction. In order to prevent the start of a 32 byte bundle being the middle of a valid instruction NaCl enforces the additional constraint that an instruction cannot cross the border of a 32 byte bundle. Instead, the former bundle must be padded with a NOP sled (a sequence of no-operation instructions) and the instruction inserted at the start of the next bundle to follow. These constraints enable the validator to reliably disassemble untrusted code and assert that the executable conforms to the subset of legal instructions [YSD+09, p. 4 and 5].

In order to ensure that all indirect jumps target the start of a 32 byte bundle, the validator requires the compiler to replace all indirect jumps with the nacljmp pseudoinstruction shown in Listing 2.8. Untrusted code cannot interpose a *nacljmp* because the entire operation is contained within

[8] *XFI.*,http://pdos.csail.mit.edu/6.828/2006/lec/l-xfi.html, last accessed 10/08/2015

a single 32 byte bundle.

```
and %eax, 0xffffffe0
jmp *%eax
```

Listing 2.8: The nacljmp instruction clears the lower 5 bits of the address to enforce 32 byte bundle alignment [YSD+09].

The nacljmp pseudoinstruction in Listing 2.8 performs a bitwise `AND` on the target address in register `%eax` with the literal value `0xffffffe0` and stores the result in `%eax`. This bitwise `AND` clears the lower 5 bits which forces the 32 byte bundle alignment (since 5 bits can represent the numbers between 0 and 31).

In addition, NaCl disables opcodes such as `syscall`, `int` and `ret` from within untrusted code. Instead, all privileged operations, such as system calls, must be mediated via a thin layer of trusted code known as the service runtime. The service runtime checks whether the system call is allowed from untrusted code and whether its arguments are safe.

In order to constrain jumps, loads and stores to their respective fault domain NaCl uses the 80836 segment registers present in x86. However, these registers are absent in x86-64. Instead, NaCl utilizes the large 64-bit address space and relative addressing modes in order to provide the same isolation on x86-64.

2.7.2 Modifications for x86-64

One of the numerous advantages of x86-64 over x86 is its 64-bit address space which is over 4 billion times larger than the equivalent 32-bit address space available on x86. NaCl utilizes this large 64-bit address space in combination with a small subset of controlled addressing modes in order to provide fault isolation with little or no address sandboxing at all.

For each untrusted module Native Client maps a contiguous 84GB subrange of the 64-bit virtual address space. The middle 4GB of this subrange is reserved for the untrusted modules program text, initialized and uninitialized global and static variables, stack and heap. Together, these represent the untrusted modules address space. Untrusted code itself is compiled in ILP32 mode [SMB+10, p. 2] such that int, long and pointer are all 32-bits in untrusted code. The remaining 80GB is split into two 40GB guard zones which flank the 4GB address space on either side. Each guard zone consists entirely of unmapped, protected pages which enables untrusted modules to

use a number of addressing modes with little or no sandboxing at all.

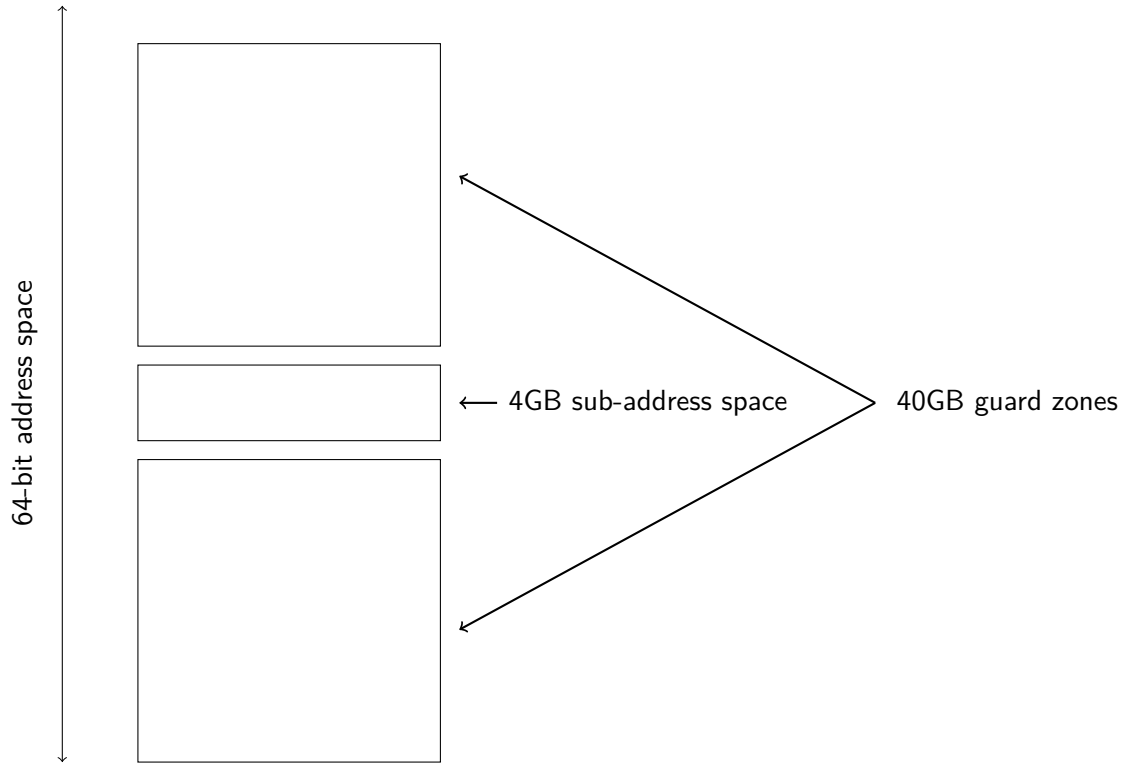

Figure 2.1: For each untrusted module NaCl maps a contiguous 84GB subrange of the 64-bit virtual address space. The middle 4GB is reserved for the untrusted module's address space. The remaining 80GB is split into two 40GB guard zones which flank it on either side.

In order to sandbox indirect jumps and support non-sandboxed addressing modes NaCl reserves a designated 64-bit register called `%RZP`. This designated register holds the base address of a NaCl module's 4GB address space and cannot be modified from untrusted code. Since the address space of each NaCl module is 4GB in size (the same limit as with a 32-bit address space) the upper 32 bits of each address are reserved for the base address of the 4GB address space: much like the segment identifiers present in the original SFI implementation [WLAG93].

As with NaCl on x86 and the implementation of SFI for MIPS [WLAG93], all direct jumps can be checked statically by the validator. However, indirect jumps must be replaced with a sequence of sandboxing instructions (see Listing 2.9) which cannot be interposed due to the 32 byte bundle

alignment.

```
and %eax, 0xffffffe0
lea (%RZP, %rax, 1), %rax
jmp *%rax
```

Listing 2.9: The nacljmp instruction for x86-64 clears the lower 5 bits of the address to enforce 32 byte bundle alignment [YSD+09]. It then adds the base address of the 4GB sub-address space stored in %RZP to the masked 32-bit address stored in the 64-bit register %rax to derive a valid 64-bit address that is within the fault domain. This address is then stored in destination register %rax which can be jumped to in the instruction that follows.

Listing 2.9 shows the sandboxing instructions for indirect jumps. Like the nacljmp in Listing 2.8 for x86, the first instruction clears the lower 5 bits of the address to enforce 32 byte bundle alignment. However, unlike on x86, registers on x86-64 are 64 bits wide. Therefore, %eax corresponds to the lower 32 bits of the 64-bit register %rax. Thus, the operation on the first line of Listing 2.9 has the additional function of clearing the upper 32 bits of the 64-bit address which denote the base address of the untrusted code. This prevents untrusted code from breaking out of the fault domain through modification of these upper 32 bits. The second instruction then sets the upper 32 bits of the address to the base register for NaCl module's address space %RZP. At last we can jump to the sandboxed address.

The 4GB address space is flanked with 40GB guard zones in order to enable sandbox-free displacement addressing for load and store operations which has a significant improvement on performance. A displacement is a special mode of addressing that takes a base address with an offset, scalar and displacement. With a maximum scalar of 8, an untrusted displacement can address a maximum address of $\%RZP + ((2^{10} * 4) * 8) + 31$ and is well within the upper 40GB guard zone. An untrusted module that attempts to deference an addresses in either the upper or lower guard zone will suffer a segmentation fault.

This method of fault isolation might sound wasteful. However, the 80GB of unmapped pages per untrusted module consume no more physical memory than the memory required for their upper level page tables. Furthermore, given that 48 bits are available for addressing on most x86-64 systems, NaCl can theoretically support in excess of three million sandboxes before exhausting the virtual address space[9].

[9] $2^{48}/84 * 2^{20} = 3195660$

Note that this report uses the terms sandbox, NaCl module, NaClApp, and fault domain interchangeably. We refer to untrusted code as code within a sandbox.

2.8 Review

In this chapter we established background knowledge essential to following the remainder of this report. We started with an intensive introduction to the memory layout of C programs. In order to leverage this knowledge in the development of exploits and set the scene for the second half of our evaluation we explained the fundamentals behind buffers in C and demonstrated how an attacker can overrun a buffer in order to exploit running software.

We introduced the Principle of Least Privilege as an effective primitive to constrain exploits via the separation of software into least privilege compartments. We discussed two previous works that enforce the Principle of Least Privilege using Unix processes (and variants thereof) and described SFI: a fault isolation primitive for running least privilege compartments within the same address space (and hence the same Unix process). We then finished the chapter with an introduction to NaCl: an implementation of SFI to enable the secure execution of untrusted, x86 and x86-64 native code in the browser.

In the next chapter we discuss our efforts towards SFI for least privilege web servers.

3 | Design and Implementation

This chapter explains how we split the HTTP parser and SSL session setup modules into reduced-privilege compartments sandboxed with SFI.

We open this chapter by discussing two different isolation models that can be enforced in a reduced-privilege web server and evaluate their relative trade-offs. We refer to these two different isolation models as *soft isolation* and *strict isolation*.

Since each reduced-privilege compartment satisfies a distinct role in the request-response cycle of an HTTP server, we require that the server can pass messages back and forth between reduced-privilege compartments. This requires an inter-compartment communication protocol such as domain sockets for Unix processes and Inter-Module Communication (IMC) in NaCl. We compare the two IMC primitives that are supported in NaCl: custom descriptors and the NaCl datagram service in Section 3.3.

Before we describe how we split the HTTP parser for each of our reduced-privilege web servers - one simple, non-concurrent server and a second, HTTP standards compliant and concurrent web server - we first outline the modifications that were required of the NaCl source code in order to enable our strict isolation model, namely the addition of a sandbox destruction function to enable the continuous creation and destruction of sandboxes from within trusted code.

We first consider fault isolation for the simple web server written in the first week of our project. We describe each of the variants of the simple web server and explain how we separated the HTTP parser into a reduced-privilege compartment for both our soft and strict isolation models. We build a server variant for each isolation model: one isolated with SFI and the other isolated with separate Unix processes.

We continue with a discussion of suitable HTTP standards-compliant, concurrent web servers and justify our choice of the Monkey HTTP server. We then explain how we sandboxed the HTTP

parser for the Monkey HTTP server and built reduced-privilege implementations for both our soft and strict isolation models isolated via SFI and separate Unix processes.

Last, we describe our modifications to NaCl's compiler toolchains and program validator in order see whether disabling the addition of read sandboxing instructions for untrusted code improves web server performance.

3.1 Soft vs. strict isolation

Rather than making software unexploitable, the purpose of fault isolation is to ensure that faults, whether exploitable or not, are constrained within their respective fault domain. In the case of isolation via Unix processes, faults are constrained to the address space of the faulting process in addition to all other operations, such as system calls, that are permitted for that process. In the case of SFI, faults are constrained to the security subdomain for the distrusted module. In NaCl, this means that untrusted code cannot break outside the sandbox nor initiate privileged operations, such as system calls, which are disallowed by the mediating service runtime.

However, whilst the Principle of Least Privilege ensures that a reduced-privilege compartment cannot compromise the confidentiality nor the integrity of other reduced-privilege compartments or trusted code (such as reading and writing to addresses in their address space, modifying their control flow or executing instructions in their program text), the Principle of Least Privilege does not prevent a compromised reduced-privilege compartment from being able to read, write modify data within the compromised compartment.

This presents a number of challenges for a reduced-privilege web server. For example, if we suppose a reduced-privilege web server creates a single, long-lived reduced-privilege HTTP parser which is reused over its entire lifetime, then the compromise of the HTTP parser itself could result in an attacker being able to manipulate the HTTP requests for other clients that interact with the same server.

This need for stricter isolation between users of reduced-privilege compartments stipulates the need for two different isolation models which we refer to as *soft* and *strict* isolation.

In *soft isolation*, a reduced-privilege web server creates exactly one instance of a reduced-privilege compartment to be used for all client requests over the server's lifetime. In comparison, *strict isolation* requires that a reduced-privilege web server creates a separate instance of a reduced-

privilege compartment per client request. This ensures that even if a vulnerable HTTP parser is compromised it cannot manipulate the HTTP requests of other clients as each client request has its own one-time-use instance of each reduced-privilege compartment.

Therefore, whilst both isolation models ensure that a faulting compartment cannot compromise other reduced-privilege compartments or trusted code, it is the second isolation model alone that also enforces isolation of faulting reduced-privilege compartments between users.

3.2 SFI vs. Unix processes

In addition to the potential improvement in reduced-privilege web server performance, SFI provides stronger default isolation than standard Unix processes.

First, the invocation of all system calls initiated from within untrusted code is prohibited by default. Instead, the invocation of a system call must be mediated via a small subset of trusted code (just 64KB in size [YSD+09, p. 6]) known as the service runtime. In contrast, a standard Unix process can invoke the same system calls as trusted code.

Second, when a Unix process calls `fork` the created child process is a duplicate of its parent and therefore inherits all state, such as the set of open file descriptors, from its parent too[10]. We refer to this as the *default-allow* model since the child process can read everything from the parent process' memory image and read and write on all file descriptors open at the time of `fork`. In contrast, the creation of a sandbox in NaCl uses a *default-deny* model in which the sandbox's address space is memory mapped to a zero initialized, copy-on-write shared memory page. This behavior is achieved via setting the `MAP_PRIVATE` and `MAP_ANONYMOUS` flags in the `mmap` system call. We validate the correctness of the `mmap` manual page which states that `mmap` returns zero initialized pages via cross examination with the Linux kernel.

3.3 Inter-module communication

Since each reduced-privilege compartment satisfies a distinct role in the request-response cycle of an HTTP server, we require that the trusted code can pass messages back and forth between reduced-privilege compartments. This requires an inter-compartment communication protocol

[10] *fork*, Linux Programmer's Manual, http://man7.org/linux/man-pages/man2/fork.2.html, last accessed 30/08/15

which is referred to in NaCl as the inter-module communication (IMC). We compare the two IMC primitives that are supported in NaCl: custom descriptors and the NaCl datagram service.

In the case of our reduced-privilege web server variants isolated with SFI, the trusted code (in which the rest of the server resides) must communicate with the untrusted, reduced-privilege HTTP parser. For example, upon accepting a client connection and reading an HTTP request into a stack-allocated buffer, the trusted code must pass the buffer to the untrusted HTTP parser to process the request and to send a parsed HTTP request struct back to the trusted code. However, since the HTTP parser runs in a NaCl sandbox it can neither read or write memory outside its fault domain.

Therefore, in order to enable communication between trusted code and untrusted code, NaCl provides IMC which supports two communication primitives as described below.

3.3.1 Custom descriptors

The first communication primitive in IMC is custom descriptors. Unlike most other communication primitives, such as Unix domain sockets and the NaCl datagram service, custom descriptors utilize shared memory.

However, in order to preserve strong isolation between untrusted and trusted code, communication over custom descriptors cannot be initiated from trusted code. Rather, when the untrusted code wants to receive a message from trusted code, the untrusted code must first call the `imc_recvmsg` function. The invocation of this function then initiates a callback in the trusted code whose arguments include a pointer into untrusted memory. The trusted code can then write its message starting at the address in the pointer.

Likewise, in order to send a message to trusted code, untrusted code must first call the function `imc_sendmsg()`. The invocation of this function then initiates a different callback in the trusted code whose arguments also include a pointer into untrusted memory. However, in order to prevent trusted code from storing the pointer whose deference could be later manipulated from within untrusted code, the trusted code must first copy the data from untrusted to trusted memory after which it must take sufficient steps to sanitize the data prior to further processing.

3.3.2 NaCl datagram service

The second communication primitive in IMC is the NaCl datagram service. Unlike custom descriptors, in which communication must be initiated from untrusted code, the NaCl datagram service enables communication to be initiated at either end. This is because, unlike custom descriptors which utilize shared memory, NaCl datagrams are copied between sender and receiver much like OS level primitives such as Unix domain sockets.

In order to send a message over the NaCl datagram service both ends must instantiate structures for the message header and the destination or source of the payload, represented as a sequence of I/O vectors with a pointer and size.

To send a message, the sender, whether in trusted or untrusted code must invoke the `NaClSendDatagram` function. Likewise the receiver, whether trusted or untrusted, must invoke the `NaClReceiveDatagram` function. Both `NaClSendDatagram` and `NaClReceiveDatagrams` are wrapper functions for system calls in the service runtime which are responsible for the copying, serialization and deserialization of messages.

However, it took the project group over a week to understand how to send messages from trusted to untrusted code. Whilst we could send messages from untrusted to trusted code, we could not send messages in the reverse direction. It was only through setting the environment variable `NACLVERBOSITY` to 10 did we find that communication from trusted to untrusted code required a 16 byte header in the payload with the first four of which non-zero.

3.4 Sandbox destruction in Native Client

In order to enforce strict isolation in our reduced-privilege SFI server variants we must first test whether NaCl supports the continuous creation and destruction of sandboxes in trusted code.

However, we found that unlike the `waitpid` function for Unix processes which waits for the process to exit and release its resources[11], the equivalent `NaClWaitForMainThreadToExit` function does not release all resources associated with the sandbox. We found no other destructor function in the source code for NaCl. We attribute this to the fact that NaCl is built for the safe execution of untrusted code in the browser, and therefore a sandbox destructor function is not required since

[11] *WAIT(2)*, Linux Programmers Manual, http://man7.org/linux/man-pages/man2/wait.2.html

less than a handful of sandboxes are created over the lifetime of each process per tab[12].

Therefore, to prevent the eventual exhaustion of the 64-bit virtual address space that would result from not releasing each 84GB sub-range reserved per sandbox (see Section 2.7.2) and the eventual termination of the host OS process that would result from the accumulation of the resulting memory leaks, our strict isolation model requires that we implement a sandbox destructor function to release the virtual address space and additional resources acquired per sandbox creation.

However, whilst our destructor function returns almost all resources associated with a sandbox, we were unable to free all heap-based allocations via `malloc` in the sandbox creation functions without much wider modifications to the structure of the NaCl source code itself. Hence, with our current destructor implementation, each additional sandbox leaks just under 2KB of physical memory as determined with the Valgrind program analysis tool.

One unexpected complication of repeated sandbox creation and destruction was that the host OS process would crash after the creation of 64,000 sandboxes. We tracked the error down to a `ENOMEM` errno set in the `mprotect` system call. We found that this error was caused from a limit `max_map_count` which sets an upper bound on the maximum number of virtual memory areas a process may own[13]. We therefore increased this limit from 64,000 to unlimited.

One final obstacle to continuous sandbox creation and destruction was the presence of a race condition in the NaCl source code. In trusted code, the main thread of execution must call the `NaClWaitForMainThreadToExit` function in order to wait for the sandbox's main thread to terminate. However, this function call alone is insufficient since the main thread in trusted code must also wait for the sandbox's underlying pthread to terminate. We reported this bug to Google[14] and fixed the race condition in the implementation of our sandbox destructor function.

3.5 Reduced-privilege simple web server

Whilst our overall objective is to sandbox the HTTP parser of a concurrent web server, we did not know which concurrent web server to sandbox nor whether this objective could be achieved

[12] *Chrome's multiprocess architecture*, http://www.chromium.org/developers/design-documents/multi-process-architecture

[13] *Understanding Virtual Memory*, http://www.redhat.com/magazine/001nov04/features/vm/

[14] *Error message "NaClRemoveThreadMu:: thread to be removed is not in the table"*. Google Native Client issues, https://goo.gl/fY3Jbr, last accessed 23/08/2015

within the time frame for the project. Therefore, in order to reduce the risk of reaching the deadline without a working reduced-privilege web server we built a simple web server which we could sandbox first.

Our simple web server is less than 100 lines of C code. Its support for the HTTP specification extends little further than GET requests. Furthermore, it cannot handle concurrent requests, compression, cache headers, authentication, SSL, nor most other features expected from a web server.

However, what our simple web server lacked in features made it ideal as an initial reference implementation for a reduced-privilege web server. Furthermore, the lack of a functional debugger (GNU Debugger (GDB) or LLDB) devolved our debugging efforts to the use of `printf` statements, the built-in NaCl logging and other primitive tools such as strace.

In the implementation of our reduced-privilege simple web server we took the baseline simple server and derived four variants as described below.

3.5.1 Soft isolation with Unix processes

The first reduced-privilege variant of the simple web server is our soft isolation with Unix processes. In this version we split the HTTP parser into a separate, long-lived Unix process that is reused for all client requests throughout the server's lifetime.

At initialization, the server creates a socket pair of domain `AF_UNIX` (for Unix domain sockets) and protocol `SOCK_STREAM`. This socket pair then serves as the communication channel between the trusted code and reduced-privilege HTTP parser.

Figure 3.1: High-level architecture of the soft isolation with Unix processes server variant. The trusted web server communicates with the untrusted HTTP parser over Unix domain sockets. In the forward direction the trusted web server sends the unparsed HTTP request to the untrusted HTTP parser. In the reverse direction, the untrusted HTTP parser sends the parsed HTTP request struct back to trusted code. Both the trusted web server and untrusted HTTP parser run as separate Unix processes.

The server then invokes the `fork` system call. In the parent (which is trusted code) the server closes one of the file descriptors in the socket pair. In the child process, the server closes the other socket pair and the file descriptor for the listening socket (which would otherwise enable an exploited reduced-privilege HTTP parser to accept incoming client connections). Last, the untrusted code `chroots`, sets its `uid` to the non-privileged user `nobody`, changes its current working directory to the jail's root directly, and calls exec on the parser executable.

When the server receives a client request it reads the entire HTTP request into an 8KB buffer. The buffer is then sent to the long-lived HTTP parser process where it is parsed into an HTTP request struct. Next, the struct is sent back over the same socket pair to the trusted code which receives the parsed request struct and completes the HTTP request-response cycle. Since the simple web server is sequential and is therefore not concurrent, we do not need identifiers to distinguish separate HTTP requests.

3.5.2 Strict isolation with Unix processes

The second reduced-privilege variant of the simple web server is our strict isolation model with Unix processes. In this version we split the HTTP parser into a separate one-time use Unix process per client request (with the same compiled HTTP parser code from the soft isolation variant).

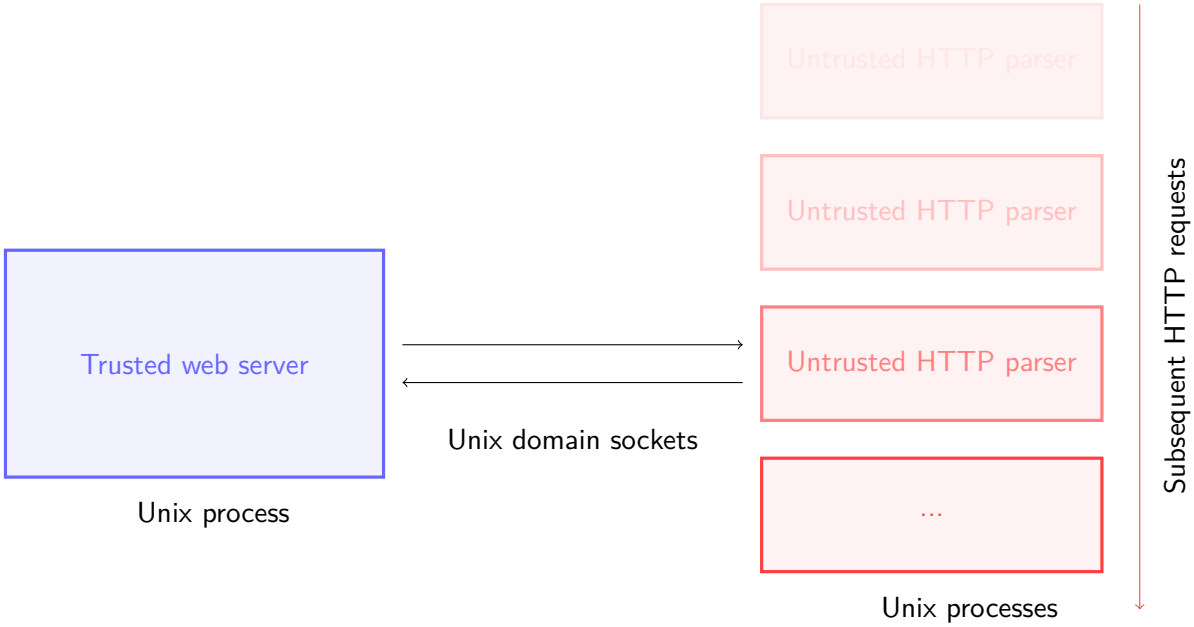

Figure 3.2: High-level architecture of the strict isolation with Unix processes server variant. The trusted web server creates a separate untrusted HTTP parser process per client request which it communicates with over Unix domain sockets. In the forward direction the trusted web server sends the unparsed HTTP request to the untrusted HTTP parser. In the reverse direction, the untrusted HTTP parser sends the parsed HTTP request struct back to trusted code.

However, rather than creating the socket pair and parser process at server initialization, our strict isolation server variant requires that we create a separate socket pair and `fork` once per client request. Like the soft isolation variant, the parent (trusted code) closes one side of the socket pair whilst the child process closes the other socket pair and listening socket, calls `chroot`, `setuid`, `chdir`, and `exec`.

Like the soft isolation variant, when the server receives the client request it reads the HTTP request into an 8KB buffer. However, the buffer is then sent to the reduced-privilege HTTP parser process distinct for that client request. The parser process then exits once the parsed HTTP request struct has been sent to the trusted code in order to free its resource and ensure that it cannot be reused for other client requests.

3.5.3 Soft isolation with SFI

The third reduced-privilege variant of the simple web server is our soft isolation model with SFI. In this version we split the HTTP parser into a separate NaCl module. On initialization, the server (which runs in trusted code) creates a single, long-lived sandbox for the NaCl HTTP parser module which it communicates with over the NaCl datagram service (see Section 3.3.2).

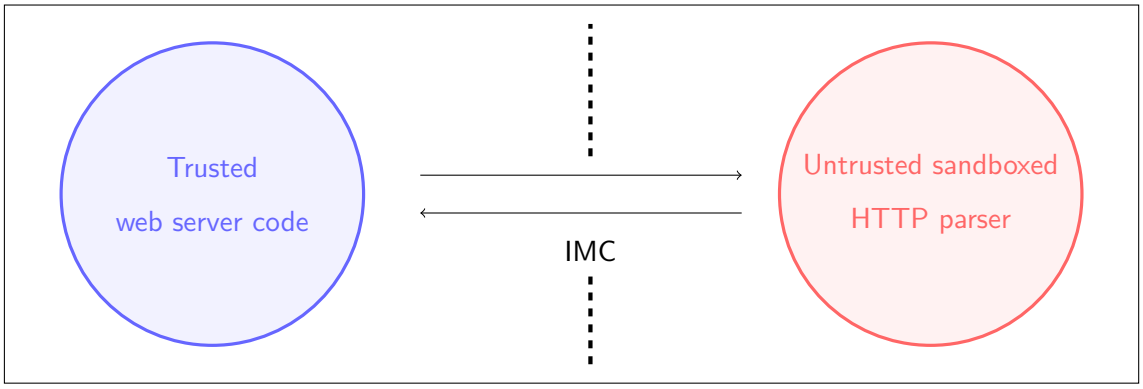

Figure 3.3: High-level architecture of the soft isolation with SFI server variant. The trusted web server code creates a reduced-privilege long-lived untrusted HTTP parser NaCl sandbox. The trusted web server code and untrusted HTTP parser communicate over IMC.

We did not use the NaCl custom descriptors in this variant since custom descriptors require that communication is initiated from untrusted code. This would require the HTTP parser to know when the server had received a client HTTP request. Whilst we could have implemented long polling in the parser it would have degraded performance when compared to communication via the NaCl datagram service.

There is no need to `chroot` the NaCl module or `setuid` to `nobody`. Instead, the service runtime mediates all privileged operations, including system calls, that can be invoked from within untrusted code.

3.5.4 Strict isolation with SFI

The last reduced-privilege variant of the simple web server is our strict isolation model with SFI. In this version we use the same NaCl HTTP parser module as in our soft isolation SFI server variant. However, rather than creating a single, long-lived sandbox on server initialization, the

server creates a separate, single-use HTTP parser NaCl sandbox per client request. This ensures that an exploit of one HTTP parser sandbox instance cannot manipulate the HTTP requests for other clients that interact with the same server.

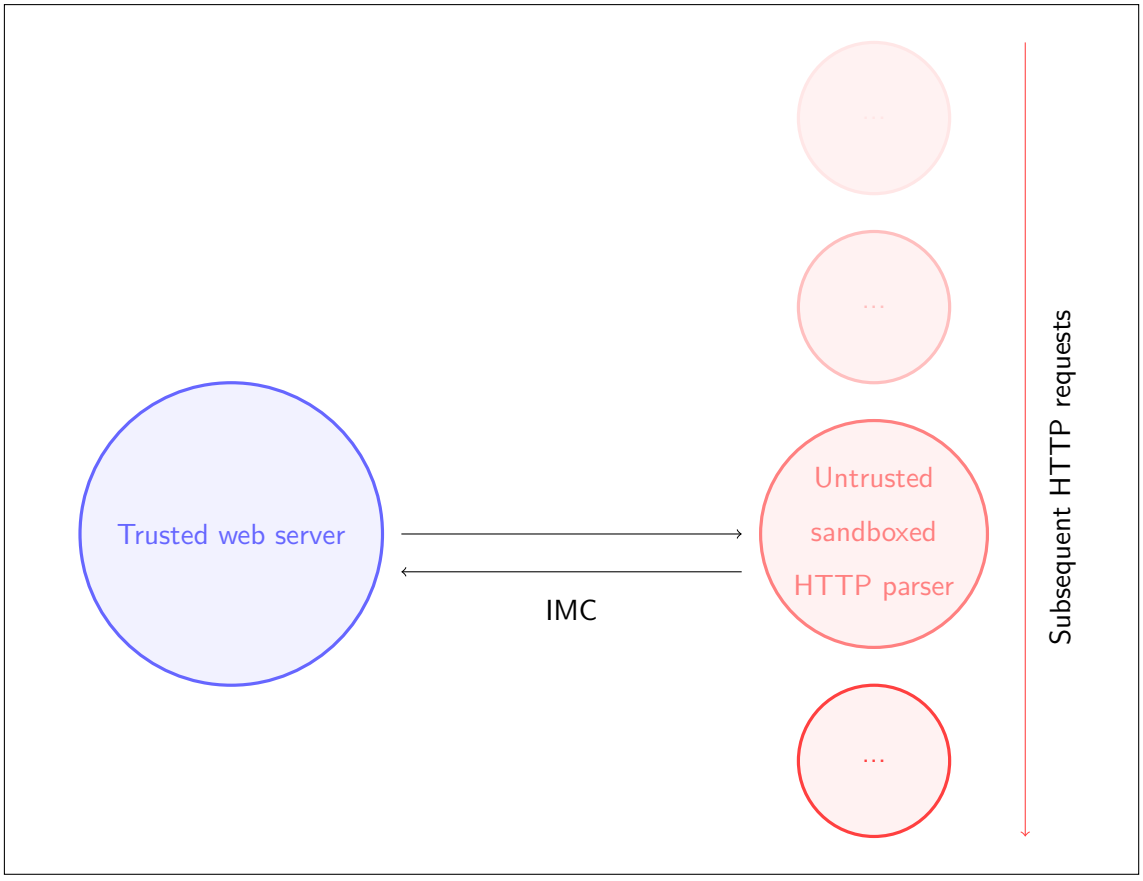

Figure 3.4: High-level architecture of the strict isolation with SFI sandboxes server variant. The trusted web server creates a separate untrusted HTTP parser NaCl sandbox per client request which it communicates with over IMC. Each HTTP parser NaCl sandbox instance is never used more than once. When the trusted web server has finished sending the HTTP response to the client it calls the `NaClAppDtor()` function to destroy the sandbox.

3.6 Reduced-privilege concurrent web server

With the simple web server and our sandbox destructor function for NaCl complete, we initiated a discussion over which concurrent web server we should sandbox with SFI. Whilst we had first intended to sandbox either the Apache HTTP server or the Nginx HTTP server, it was a question

from Prof. Mark Handley in the Q&A of our project presentation that forewarned about their intrinsic complexities and whether it was attainable within the three month time frame which led us to reconsider.

Therefore, we set out to find an open-source, concurrent HTTP server implementation with all the features expected of an HTTP server implemented with the fewest lines of C code. Our search led us to two lightweight, concurrent web servers: lighttpd and the Monkey HTTP server. However, whilst lighttpd had seen real production use at YouTube[15] and other high-traffic websites, we concluded that it was still too complex for the time frame of our project and would otherwise constitute as additional risk. In contrast, the Monkey HTTP server supported all the features required for our purposes, such as HTTP 1.1 compliance and an event-driven, multi-threaded execution model in 14146 lines of code[16]: a 427% reduction over lighttpd[17].

However, unlike in the simple web server, the parsing of the HTTP request buffer takes place at different positions throughout the Monkey HTTP server. Hence, whilst the HTTP method is parsed immediately after reading the HTTP request from the socket, the remaining header fields are parsed just prior to the method invocation that writes the HTTP response back. Therefore, in order to compartmentalize the Monkey HTTP server, trusted code must communicate with the untrusted parser multiple times over IMC to process each different part of the HTTP request.

In addition, the Monkey HTTP server makes extensive use of pointers when parsing an HTTP request. The parser creates a separate struct, known as a `mk_ptr`, with a pointer to the start of each field in the HTTP request and its length in bytes.

However, this does not work for reduced-privilege web servers. First, in the case of isolation via *Unix processes*, a reduced-privilege HTTP parser cannot read across process boundaries and therefore cannot create pointers that point into the original request buffer stored in trusted code. Likewise, in the case of isolation via *SFI*, a reduced-privilege HTTP parser cannot reference addresses outside its sandbox and therefore cannot create pointers that point into the original request buffer stored in trusted code either.

On the other hand, in the case of isolation via SFI, untrusted code can send pointers to trusted code that point back into the untrusted code's address space since trusted code can reference the full 64-bit process address space. Yet, whilst this appears to be an ideal solution, it in fact poses

[15] *Youtube Architecture*, http://highscalability.com/youtube-architecture, last accessed 30/08/15

[16] `find src | xargs wc -l`

[17] $60483/14146 = 4.276$

a number of additional hazards. For example, suppose that the HTTP parser in untrusted code sends a list of pointers to char to trusted code over IMC, one for each parsed HTTP request field. The trusted code then validates that each of the pointers points to an expected address within the untrusted code's address space and that each of the strings pointed to are safe. However, since the pointers point into the address space of untrusted code, the untrusted code can manipulate any number of these strings between validation and further processing in the trusted code.

To avoid such vulnerabilities, we instead require that untrusted code sends a list of tuples, each of which consists of an offset from the start of the HTTP request (in bytes) to the respective HTTP header field and a length which denotes the length of the field in bytes. Trusted code can then translate these offsets back into `mk_ptr` structs.

However, there is one last complication that we have yet to consider. NaCl modules are compiled in ILP32 mode [SMB+10, p. 2] whilst trusted code is compiled in the standard ILP64 mode for x86-64. This means that for the transmission of the parsed HTTP request from untrusted to trusted code, we had to ensure that our message structs contained platform-independent padding and alignment in addition to the use of consistent primitive data types between modes. We plan to use platform independent, width explicit primitive data types such as `int32_t` in future.

We implement each of the four server variants: soft isolation with Unix processes, strict isolation with Unix processes, soft isolation with SFI and strict isolation with SFI like in the simple web server described in Section 3.5.

3.7 Reduced-privilege SSL module

The Monkey HTTP server uses the open-source mbedTLS[18] (formerly known as polarssl) SSL/TLS implementation to enable communication over secure connections between client and server. mbedTLS contains networking components which form part of the server's transport layer, a key-pair generation module, a cipher module for encryption and decryption and an X.509 module for SSL certificates.

As described in Section 1.1, the SSL module is tasked with both the setup of an SSL connection between client and server in addition to the encryption and decryption of messages for established SSL sessions. However, even if the SSL module is rid of vulnerabilities, both the SSL private

[18] *mbedTLS*. ARM issues, https://tls.mbed.org/, last accessed 16/08/2015

key and confidential SSL session information can still be compromised since the SSL module runs within the same address space as the rest of the server. This means that an exploit of a vulnerability outside the SSL module could compromise a secure, well-audited and vulnerability-free SSL module because of the monolithic memory model where all server components run with full privileges in the same process address space.

Note that this design to protect the SSL module is different from the prior designs to protect the server from a potentially faulty HTTP parser and takes a completely different threat model as basis. Whilst we now assume that the SSL module is in our TCB, the remaining server may contain exploitable vulnerabilities.

Hence, we propose running the entire web server as a NaCl module (and therefore sandboxed) with the SSL module running in trusted code. Thus, even though the remainder of the server can read and write to established network connections, it cannot read or write memory from the SSL module since it lies outside the fault domain in trusted code.

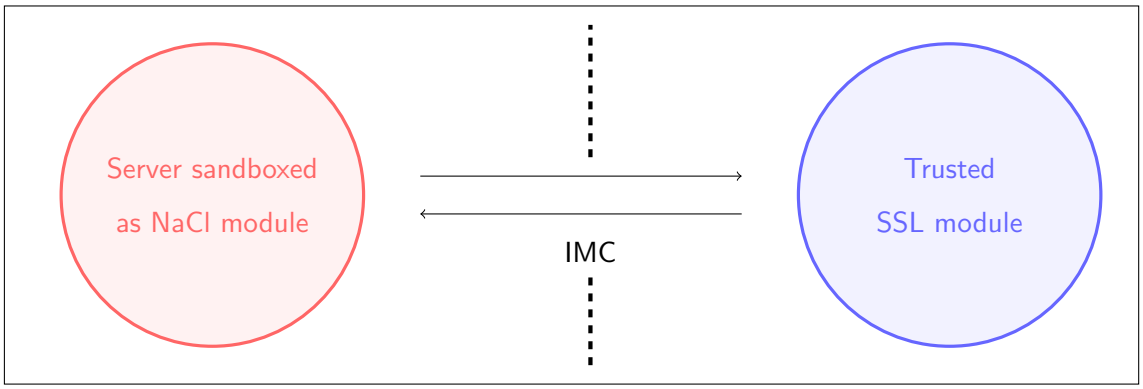

Figure 3.5: The high-level architecture for protecting the SSL module. The server and all other functional components run in a unified untrusted NaCl module. The reduced-privilege server and trusted SSL module then communicate over IMC.

However, the service runtime in NaCl blocks the invocation of socket related system calls, such as `accept()`, `bind()` and `listen()` from untrusted code [YSD+09, p. 4]. Therefore, in order to run the web server as a NaCl module we needed to hack the service runtime in order to support these additional system calls.

That said, we were unable to complete this task within the time that remained. The modifications to NaCl required to support additional system calls took much longer than expected. We ran

into frequent obstacles with missing Linux kernel headers and missing functions in NaCl's libc implementation. Furthermore, whilst our simple web server requires just four additional socket related system calls, the Monkey HTTP server requires more than a dozen. Therefore, we exclude the reduced-privilege SSL module from our evaluation and reserve its continuation for future work.

3.8 Supporting reads outside the sandbox

In order to prevent untrusted code from reading and writing data and executing code outside its fault domain NaCl's compiler toolchain, as described in Section 2.7, inserts sandboxing instructions around indirect jumps, loads and stores which force the address within the 4GB sub-address space and two 40GB guard zones which surround it.

We attempt to remove the insertion of these read sandboxing instructions in the NaCl compiler toolchains. However, we found that NaCl's isolation model requires the x32 Application Binary Interface (ABI) for x86-64 such that untrusted code is compiled in ILP32 mode. Hence, primitive data types such as `int`, `long` and `pointer` are all 32 bits wide. As a result, all untrusted code compiled with the NaCl toolchain is constrained to a 32 bit wide 4GB sub-address space. The same is not true of hand-written x86-64 assembler.

Since trusted code requires a 64-bit address space in order to map the 4GB sub-address space and two 40GB guard zones for each instance of untrusted code, known as a sandbox, the NaCl compiler toolchain outputs two different binaries: one 64-bit for the trusted code and one ILP32 mode for the untrusted code.

The code generators for both NaCl's GCC and LLVM based compiler toolchains translate their intermediate codes (Register Transfer Language (RTL) on GCC and intermediate-representation (IR) on LLVM) into architecture dependent machine code. This is where each compiler toolchain inserts sandboxing instructions required to enable the safe execution of untrusted code. These additional instructions ensure that all updates to the instruction pointer are replaced with the `nacljmp` pseudo-instruction (shown in Listing 2.9) and that all loads and stores which do not use a 32-bit displacement from the stack pointer, base pointer or dedicated base register `%RZP`, such as when dereferencing a pointer, require instrumentation via a sequence of masking instructions.

```
mov %eax, %eax
mov (%r15, %rax), %eax
```

Listing 3.1: Example masking instruction that clears the upper 32 bits of a 64 bits register.

In the masking sequence shown in Listing 3.1, the first instruction clears the top 32 bits in the register %rax with a mov operation on %eax to itself, as explained in [SMB+10, 3.2]. The second instruction then adds the base for the 4GB sub-address space, stored in register %r15 (equivalent to %RZP) in order to produce a valid 64-bit address that points within the untrusted code's 4GB sub-address range. However, whilst Listing 3.1 has a scale factor of 1, and is therefore omitted, even with a maximum scale factor of 8 the 64-bit address is still forced within the 40GB guard zones on either side.

Changing the compiler toolchain to compile untrusted code with 64 bits would break the limitations enforced with displacement addressing (see Section 2.7.2) and thus enable untrusted code to both read *and* write outside of the sandbox. We have not been able to implement an alternative isolation scheme which would enable the safe execution of untrusted 64-bit compiled executables. We believe that such an implementation would significantly reduce the performance of untrusted code on x86-64 as it would require *all* load and store instructions to be sandboxed (rather than just the set of instructions that do not use displacement addressing).

Nevertheless, we remove the masking instruction shown in line 1 of Listing 3.1 as they are excessive given the ILP32 bit data model used in untrusted code. Furthermore, we find that read sandboxing was implemented in NaCl in order to protect sensitive data structures accessible at well-known addresses in Windows[19]. We therefore modify both the GCC and LLVM compiler toolchains to prevent the insertion of these masking instructions and modify the NaCl validator to skip checking for the presence of these masking instructions.

```
// case SEG_NACL:
    // case SEG_NACL:
        // fputs ("nacl:", file);
        // break;
```

Listing 3.2: We removed the insertion of the "nacl:" prefix for segment registers which prevents the addition of masking instructions in GCC's code generator.

[19] *NaCl compiler source files*, https://goo.gl/3YsxWO

Listing 3.2 shows our modifications to the `print_operand_address_parts` function in `gcc/gcc/config/i386/i386.c` for the NaCl GCC compiler. In particular, we comment out the case statement that prepends the label `nacl:` which instructs the code generator to insert additional masking instructions[20].

```
// unsigned Reg32 = getX86SubSuperRegister_(Reg64, MVT::i32);
// EmitMoveRegReg(STI, false, Reg32, Reg32, Out);
```

Listing 3.3: We commented out the addition of masking instruction that clear the upper 32 bits in 64 bit registers.

Likewise, Listing 3.3 shows our modifications to the `EmitRegTruncate` function in `llvm/lib/Target/X86/MCTargetDesc/X86MCNACL.cpp` for the NaCl LLVM compiler. The first line of Listing 3.3 extracts the identifier for the sub 32-bit register from its 64-bit wide counterpart. The second line then inserts a `mov reg, reg` instruction in order to clear the upper 32 bits in the 64-bit register. We comment out both lines in order to disable the addition of these masking instructions.

```
...
status = NaClValidationSucceeded; // Force validation to pass
return status;
```

Listing 3.4: The modification we made to the NaCl validator to remove checks for masking instructions to enable execution of untrusted code compiled with the modified compilers.

Last but not least, Listing 3.4 shows the modifications to the `ApplyDfaValidator_x86_64` function in `trusted/validator_ragel/dfa_validator_64.c` for the NaCl validator. Line 1 sets the value of `status` to `NaClValidationSucceeded` in order force the validator to accept untrusted code that is absent of read masking instructions.

[20] *NaCl SFI model on x86-64 systems*, https://developer.chrome.com/native-client/reference/sandbox_internals/x86-64-sandbox#x86-64-sandbox

4 | Evaluation

In this chapter, we look to validate our hypothesis that enforcing the Principle of Least Privilege with SFI yields better performance than equivalent fault isolation with Unix processes. In particular, we evaluate the performance of the simple web server and the Monkey HTTP server for both soft and strict compartment isolation (see Section 3.1) when compared with their baseline and process isolated counterparts.

We expect the performance for both the simple web server and the Monkey HTTP server with soft SFI isolation between reduced-privilege compartments to be worse than their baseline counterparts but better than strict isolation between compartments enforced via Unix processes.

However, in order for the strict isolation variants of our simple web server and the Monkey HTTP server to have better performance with SFI (which creates a separate sandbox per client request) than with Unix processes (which `fork/exec` to create one reduced-privilege process per client request) either of the following potential outcomes must hold true:

1. Sandbox creation is faster than `fork/exec` and the use of IMC for message passing is at least as fast as inter-process communication.
2. Sandbox creation is slower than `fork/exec`. However, the overhead of inter-process communication for Unix processes is sufficient to offset the initial cost of sandbox creation.

In order to establish whether either of these potential outcomes holds true and therefore expect our strict isolation SFI variants to outperform their Unix process counterparts, we set up a number of microbenchmarks to first compare the performance between sandbox creation and `fork/exec` and second to compare the performance between IMC and IPC.

With our microbenchmarks complete we then set up a number of macrobenchmarks to measure the baseline performance of our simple web server and the Monkey HTTP server. Next, we compare their baseline performance with their reduced-privilege SFI and process isolated variants

for both our soft and strict compartment isolation models.

In the remainder of this chapter, we evaluate the isolation of sandboxed untrusted code in which we show that untrusted code sandboxed with SFI cannot read, write or execute outside its fault domain. To demonstrate that untrusted code cannot read or write outside its fault domain, we pass a pointer from trusted to untrusted code via IMC and attempt to dereference the pointer in order to read and write at the address. We then examine the sandboxing instructions which force the address stored in the pointer to point within the untrusted code's 4GB address space. To show that untrusted code in a reduced-privilege compartment cannot execute trusted code we first exploit the HTTP parser of our baseline simple web server via a malicious HTTP request. The HTTP request overflows a buffer on the stack (see Section 2.3.3) in order to overwrite the return address and jump to a privileged function. We then demonstrate that the equivalent exploit for the sandboxed HTTP parser cannot jump to the same trusted code even though it resides within the same process address space.

Last, we evaluate the performance of untrusted code having disabled the addition of masking instructions in NaCl's compiler toolchains and we conclude with a discussion of our results.

4.1 Microbenchmarks

Whilst reusing reduced-privilege compartments provides good performance [BMHK08, p. 4] [Kro04, p. 4] it also weakens isolation between users. Therefore, in order to enforce strong isolation we must create a separate instance of a reduced-privilege compartment per client request. Whilst we can create a separate NaCl sandbox per compartment per client request, a similar effect can also be achieved with the `fork` and `exec` system calls in which the server forks a new reduced-privilege process for each compartment per client request. However, `fork` and `exec` provide weaker isolation than SFI since the programmer must remember to isolate the child process (i.e. close inherited file descriptors, chroot, set uid and gid, disable privileged syscalls with SELinux, etc). We therefore look to compare the performance of sandbox creation with `fork/exec` to see whether we can achieve stronger isolation at an equal or better performance.

In addition to the costs of reduced-privilege compartment creation and destruction, the coordination of reduced-privilege compartments requires some means of communication. We hypothesize that IMC is faster than IPC due to the following reasons:

1. Unlike IPC, IMC does not need to trap to the kernel since all compartments run within the same Unix process.

2. IMC does not incur the overhead of copying messages to the kernel and back since all compartments run within the same Unix process.

We run all microbenchmarks on a Lenovo IdeaPad Y510p with an Intel(R) Core(TM) i7-4700MQ CPU 2.40GHz processor, 4 physical (8 virtual) CPU cores, 8GB RAM and running Ubuntu 14.04.

4.1.1 Process vs. NaCl sandbox creation

In the first microbenchmark we looked to compare the cost of `fork/exec` with that of NaCl sandbox creation using two simple programs. The first program sits in a tight loop invoking the `fork` system call. The child process calls `exec` whilst the parent calls `waitpid`. Each iteration prints out the execution time for the process creation and termination measured with `clock_gettime` nanosecond timers.

The second program is similar to the first except that it creates an instance of a NaCl module rather than a separate Unix process. Both load the same executable: a C program with a single `return 0` statement as its main function. We run each program five times with 800 iterations of the tight loop in each.

However, in order to create multiple NaCl sandboxes within the same Unix process we had to implement our own destructor function `NaClAppDtor` that would release a sandbox's resources upon termination (see Section 3.4). To present a fair comparison, we include the destructor in all sandbox versions of Figure 4.1 since, in a similar manner, a Unix process which performs a wait "allows the system to release the resources associated with the child"[21] too. Figure 4.1 shows our results.

The vertical axis in Figure 4.1 depicts the joint creation and termination times in microseconds. The green bars represent the mean values while the red bars depict the 99th percentiles.

The two programs aforementioned correspond to the first two bars `fork` and `non optimized sandbox creation` as labeled on the horizontal axis. Recall that `fork` creates and executes a new Unix process and then waits for its termination whilst `non optimized sandbox creation` creates a new NaCl sandbox, waits for the termination of the sandbox's main thread and then

[21] *man pages for wait()*. Linux programmer's manual

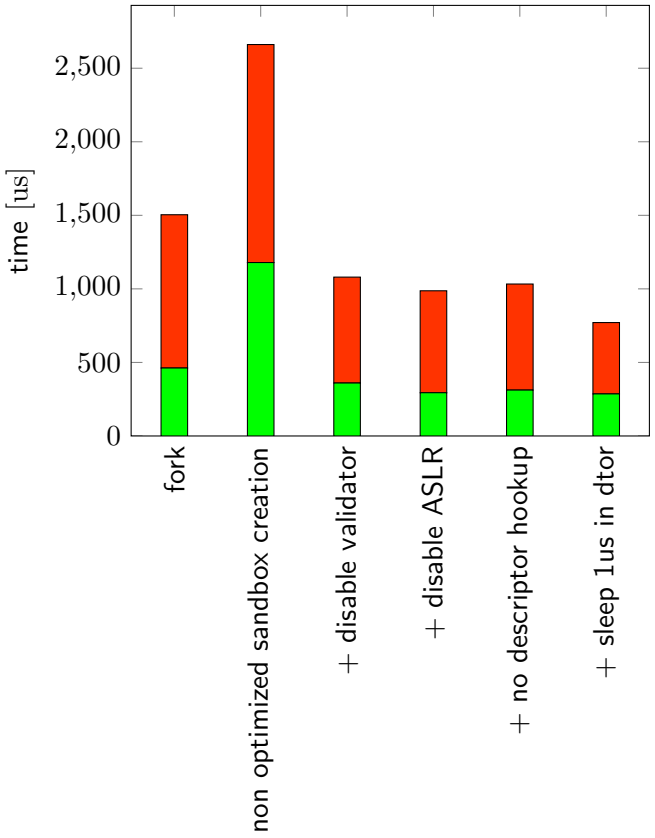

Figure 4.1: Fork vs. NaCl sandbox creation times

releases its resources.

As shown in Figure 4.1, the mean creation and destruction time of a NaCl sandbox is 2.5x that for a Unix process. One explanation for the poor comparative performance of NaCl sandboxes is that on creation of a NaCl sandbox the NaCl validator must check whether all unsafe instructions in the .nexe executable have been correctly sandboxed. However, whilst validation is required for the safe execution of untrusted native code from remote servers in the browser, in our use case, there is no need to assert whether the untrusted HTTP parser has the correct sandboxing instructions. This is because we trust the compiler to insert the correct instrumentation instructions at compile time. Furthermore, our threat model (see Section 1.3) assumes that an attacker cannot modify the compiled object code of an untrusted NaCl module. We propose modifications to NaCl which enable both fast sandbox creation and the distrust of the compiler through validation of compiled untrusted code in Future Work (see Section 6.1).

Therefore, in order to improve the performance of sandbox creation we disabled the validation of untrusted code entirely. This required that we replace the call to `subret = NaClValidateImage(nap);`

on line 397 of `src/trusted/service_runtime/sel_ldr_normal.c` with `subret = LOAD_OK`.

Our microbenchmark (see *+ disable validator* in Figure 4.1) shows that validation is the single most-expensive operation in sandbox creation. By disabling the validator we see that the cost of sandbox creation decreases 3.2x. As a result, sandbox creation without validation of the untrusted .nexe file proves to be 1.28x faster than `fork/exec`.

To see if we could improve performance further we looked to see whether we could optimize sandbox creation even more. We saw a minor performance improvement when we disabled ASLR in NaCl (which selects a random base offset for the sandbox's 4GB sub-address space within the host Unix process) and removed the call to the `NaClAppInitialDescriptorHookup` method required for file redirection of the outer sandbox [YSD+09, p. 3] when executed in the browser. Since no outer sandbox (i.e. process boundary between modules) is present in our web server, we omit this method. As shown in Figure 4.1, these two optimizations contribute a minor improvement in performance for sandbox creation. Note that all optimizations in Figure 4.1 from left (`not optimized sandbox creation`) to right (`sleep 1 us`) are cumulative, for example *+ disable ASLR* has validation disabled.

The last optimization (`sleep 1 us`) decreases sandbox creation time to a mean value of 286 microseconds. Our self-implemented sandbox destructor starts with a spin lock which waits for the destruction of the sandbox's main thread before it resumes execution. Without this lock, undesired race conditions occur where the `NaClAppDtor` function attempts to delete data structures associated with the sandbox when the sandbox has not yet released all of its resources.

In our first implementation of `NaClAppDtor`, the function would sit in a while loop and wait for the sandbox's main thread to release its resources with a sleep time of 100 microseconds between iterations. In our optimization we decreased the sleep time to 1 microsecond. This decreased the overall execution time for sandbox destruction at the expense of additional CPU cycles.

4.1.2 Inter-process communication vs. inter-module communication

In our second microbenchmark we compare the cost of IPC with IMC. In particular, we measure the performance of Unix domain sockets and compare it with that of two IMC primitives: NaCl datagrams and NaCl custom descriptors as described in Section 3.3.

For each communication primitive we sent and received 200 back-to-back messages. We repeated each run three times in order to account for the variabilities that result from scheduling and

interrupts for other processes. Figure 4.2 plots our results. It compares the mean (green bars) and 99th percentile (red bars) latencies for the round-trip times of IPC over Unix domain sockets with that of IMC over NaCl datagrams and IMC over NaCl custom descriptors.

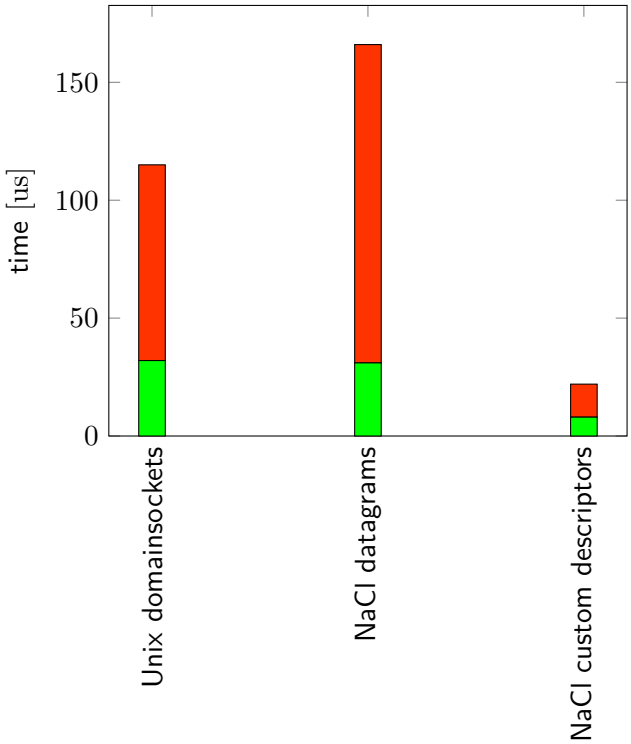

Figure 4.2: Performance comparison of IPC and IMC

We observe that the mean round-trip time for send and receive with NaCl custom descriptors is 4x less than that of Unix domain sockets and NaCl datagrams. This result confirms our expectation that custom descriptors are the fastest of the three communication mechanisms. First, when compared with Unix domain sockets, custom descriptors avoid the additional message copies required between the sending process and the kernel buffer, and between the kernel buffer and receiving process. Furthermore, Unix domain sockets require expensive context switches between processes, and the `send` and `recv` system calls require a trap to the kernel. Second, when compared with NaCl datagrams, the callbacks for custom descriptors receive a pointer into untrusted code rather than copying messages between the two.

However, even though sending and receiving messages within the same address space does not require a context switch, unlike in communication between two processes, our results in Figure 4.2 show that NaCl datagrams are slower than the equivalent communication over Unix domain

sockets. This disproves our hypothesis in which we stated that inter-module communication over NaCl datagrams is cheaper than inter-process communication between two Unix processes. We believe that this result is due to the overhead of the additional checking instructions in the `NaClSendDatagram` and `NaClReceiveDatagram` wrapper functions and the comparatively high cost of thread switches[22] between NaCl modules and trusted code.

4.2 Macrobenchmarks

With our microbenchmarks complete we now turn to our macrobenchmarks in which we measure the baseline performance (throughput in replies/sec and connection life time in ms) of both our simple web server and Monkey HTTP server. We compare each server's baseline performance with their soft and strict isolation reduced-privilege counterparts isolated with SFI and separate Unix processes.

4.2.1 Experimental setup

In the planning of the experimental setup for our macrobenchmarks it was decided that we would measure the sustained throughput (in replies/second) and connection life time of each server variant in respect to increasing offered load. To ensure that we could maintain the offered load on the server we ran a number of initial benchmarks with the ApacheBench tool, the de facto load generation tool bundled with almost all *nix distributions. However, we found that ApacheBench was ill suited for this purpose and was unable to generate load at a fixed rate in requests per second. We therefore looked at the performance benchmarks conducted in past publications and found an alternative open-source load generation tool, httperf, which could do just that.

To collect some initial results we set up an isolated test bed which consisted of one server and three load generators, all connected via gigabit Ethernet. However, even with seven out eight virtual CPU cores on the server disabled, our three client load generators were unable to saturate the server's single remaining CPU core. Hence, we needed more load generators in order to conduct our evaluation.

With just four machines at our disposal and the UCL test bed of service, we ran all macrobench-

[22] *How long does it take to make a context switch?*, http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html, last accessed 31/08/2015

marks in a cloud-based virtual environment where we could scale the number of virtual machines as needed. In all we rented eleven virtual machines, one server and ten load generators. The ten load generators were required in order to sustain a maximum attainable throughput of up to 10,000 requests/second since the httperf man page states "the maximum rate a load generating client can sustain is at most 1,000 requests per second"[23]. Each virtual machine ran Ubuntu 14.04 on a single virtual CPU core (Intel(R) Xeon(R) CPU E5-2630L v2 @ 2.40GHz) with 15 MB of L3 cache and 512MB of RAM (with 1GB for the server), inter-connected via gigabit Ethernet with private local-area networking enabled.

Whilst we could not ensure that all virtual machines resided in the same rack or even the same cluster, we did ensure that all virtual machines were in the same data center. To provide some assurance that the internal network was not the bottleneck we ran a short iperf trace to measure the bandwidth available between the server and load generators (see Appendix 9.1).

One limitation of our experimental setup was that we did not know how each of our virtual machines were provisioned, nor the ratio of virtual machines to physical servers. Furthermore, we did not know how virtual machines were scheduled in KVM nor the number of active virtual machines using the CPU at the same time. Nonetheless, we think that these benchmarks are representative of the real-world performance that can be expected from a cloud-based deployment.

All macrobenchmarks requested the same static `index.html` HTML file. Each test, unless mentioned otherwise, was repeated three times for a period of ten seconds. We would have liked to have run each test for up to 5 minutes, as recommended in the man page for httperf, however a defect in the implementation of sandbox creation resulted in frequent crashes for our strong-isolation SFI server variants which we were unable to fix in the time that remained. Whilst httperf supports HTTP keep-alive (in which TCP connections are reused for multiple HTTP requests), we did not utilize it in our benchmarking. Instead, each load generator established a separate TCP connection per HTTP request.

In order to start all ten load generators at the same time we automated our macrobenchmarks with a benchmarkrunner Python script (shown in Listing 9.2). The invocation of the benchmarkrunner (as shown in Listing 4.1) runs the benchmark over 10 load generators which connect to `10.131.8.80` on port `2001`. Each load generator starts at 25 connections/sec and finishes at 500 connections/sec in steps of 25 connections/sec, repeating each step 3 times. The coordination

[23] *httperf man page*. httperf, http://www.hpl.hp.com/research/linux/httperf/httperf-man-0.9.txt, last accessed 17/08/2015

and synchronization of each run is managed entirely by the benchmarkrunner.

```
python benchmarkrunner.py 10 10.131.8.80 2001 simple-server-baseline 25 500
    --repeat=3
```

Listing 4.1: Example invocation of the benchmarkrunner

For each load generator, the benchmarkrunner builds a unique command string for the httperf load generator tool (shown in Listing 4.2). It then creates a sub process and executes the httperf command over SSH. The output from stdout is then read back into the Python script and written to a file on the local file system. However, since untrusted code is compiled in ILP32 mode it is naturally constricted to within its own fault domain since primitive data types such as `int`, `long` and `pointer` cannot address more than 32 bits. This applies to both read and write sandboxing.

```
httperf --verbose --hog --timeout=5 --client=0/10 --server=10.131.8.80 --port
    =2001 --uri=/ --rate=25 --send-buffer=4096 --recv-buffer=16384 --num-conns
    =2500 --num-calls=1
```

Listing 4.2: Example httperf command built with the benchmark runner.

Last, for all load generators we increased the maximum number of available file descriptors and the file descriptor set size available to the `select` system call as used in httperf. To increase the number of available file descriptors we edited `/etc/security/limits.h` to increase the soft and hard limit for `nofile` to 1048576. To increase the file descriptor set size we edited `/usr/include/x86_64-linux-gnu/bits/typesizes.h` and `/usr/include/linux/posix_types.h` and set `__FD_SETSIZE` to the same limit. In addition, to prevent port number exhaustion at each load generator that resulted from TCP connections stuck in time_wait we set `/proc/sys/net/ipv4/tcp_tw_recycle` and `/proc/sys/net/ipv4/tcp_tw_reuse` to 1.

4.2.2 Simple web server

We now discuss the results from our macrobenchmarks for the simple web server. Figure 4.3 shows the sustained throughput in replies/second for each of our six different implementations of the simple web server as we increase the offered load.

The horizontal axis in Figure 4.3 represents the offered load in client connections per second. For example, at an offered load of 500 connections/sec each of the ten load generators attempts to

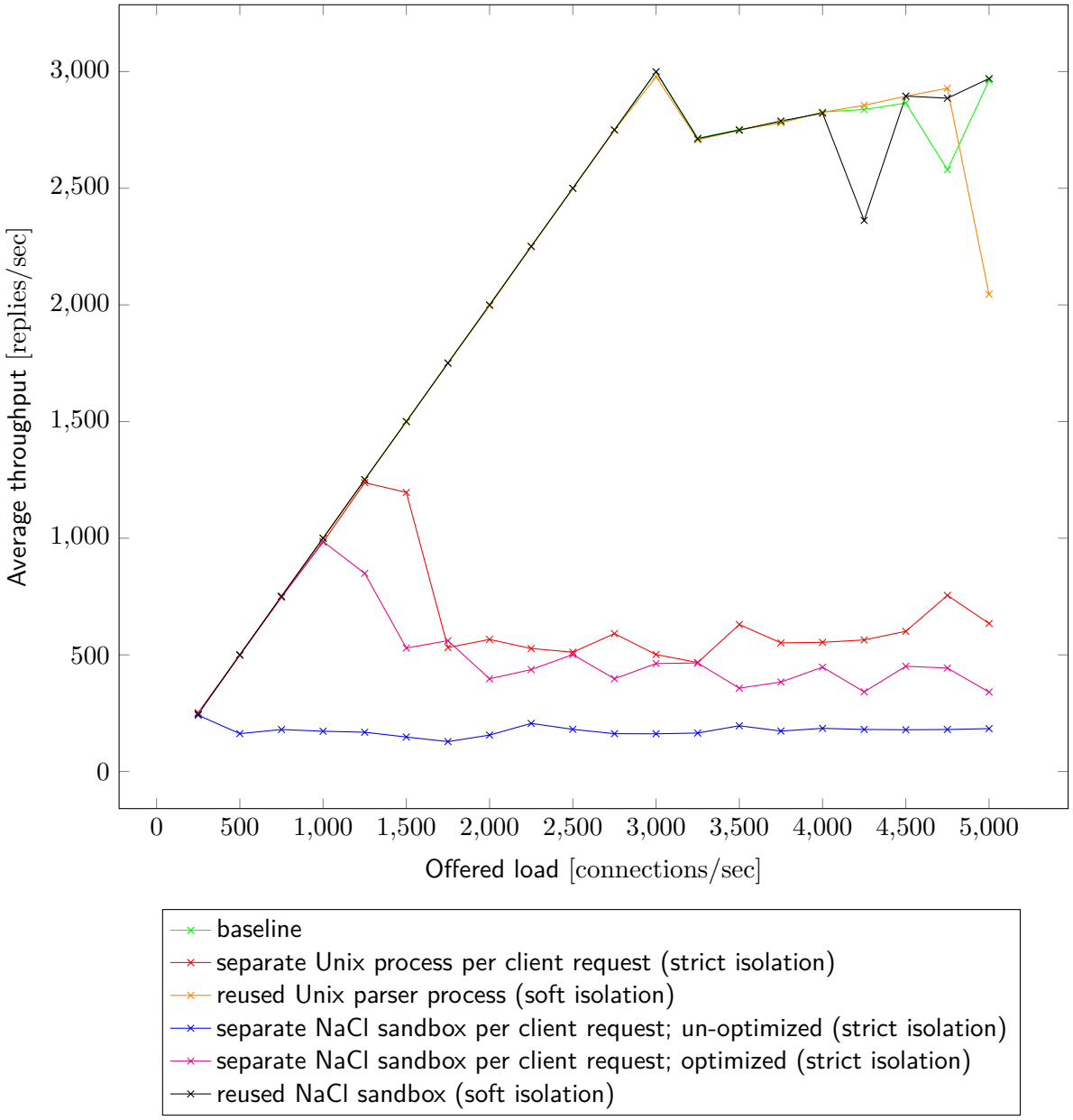

Figure 4.3: Throughput (replies/second) for the simple web server

initiate 50 TCP connections per second with the server. Note that a TCP connection might fail due to an overloaded server. In this case, the client does not receive a SYN-ACK from the server for its sent SYN message. For each established TCP connection httperf sends an HTTP request.

The vertical axis depicts the average throughput for the server in successful replies per second. For example, for an offered load of 500 connections/sec, a throughput of 500 shows that all 500 replies were received. Each point plotted in Figure 4.3 is the average over three separate test runs over a period of 10 seconds per run.

Each colored line represent a different implementation of the simple web server.

- **Green:** The baseline simple web server. The HTTP parser is not isolated.
- **Red:** The strong isolation variant of the simple web server that uses Unix processes. In this version, the server runs `fork/exec` to create a new reduced-privilege HTTP parser for each client request.
- **Orange:** The soft isolation variant of the simple web server that uses Unix processes. In this version, the server runs `fork/exec` once at server initialization. The child then initiates the HTTP parser executable. The server uses the same parser process for all client requests.
- **Blue:** The strong isolation variant of the simple web server that uses SFI. In this version, the HTTP parser is a NaCl module. The server creates a new sandbox for the HTTP parser for each client request. This version is the **unoptimized** version described in Section 4.1.1.
- **Magenta:** The strong isolation variant of the simple web server that uses SFI. In this version, the HTTP parser is a NaCl module. The server creates a new sandbox for the HTTP parser for each client request. This version is the **optimized** version described in Section 4.1.1.
- **Black:** The soft isolation variant of the simple web server that uses SFI. In this version, the HTTP parser is a NaCl module. The server creates the sandbox for the HTTP parser once at server initialization. The server uses the same parser sandbox for all client requests.

The baseline simple web server (shown in green) has the highest throughput as offered load increases. The outlier at 4,750 requests/sec is the result of inaccuracies in this run. The baseline simple server shows linear growth with $x = y$ until the peak rate of 3,000 replies per second.

The difference in throughput between the soft isolation parser process (shown in orange line) and soft isolation SFI parser (shown in black) is indistinguishable as we create just one Unix process

and NaCl sandbox at the start of server initialization. The cost of communication between the server process and the parser is negligible in comparison to the operations carried out in the rest of the server.

The performance of the non-optimized strict isolation SFI simple web server variation is worst because the untrusted code must be re-validated for each client request (see Figure 4.1 for the performance comparison of sandbox creation with and without validation).

The most interesting point in Figure 4.3 is the difference in throughput for the strong isolation implementation with Unix processes (shown in red) and the optimized strong isolation implementation with NaCl sandboxes (shown in magenta). Our results show that the Unix processes variant has better performance than the variant with NaCl sandboxes. However, our results from Figure 4.1 show optimized sandbox creation to be cheaper than `fork/exec`.

Indeed, Figure 4.3 disproves our hypothesis for the simple server. However, we find that our hypothesis is still valid for the Monkey HTTP server. For our simple server, we explain that the lesser performance of a separate NaCl sandbox per client request (compared to a separate Unix process per client request) is the result of the additional 7% [SMB+10, p. 7] runtime overhead of SFI since "NaCl module performance is impacted by alignment constraints, extra instructions for indirect control flow transfers, and the incremental cost of NaCl communication abstractions" [YSD+09, p. 8]. Due to the small amount of trusted code in the simple web server, the amount of total execution time spent in untrusted code is higher for the simple web server than it is in the Monkey HTTP server where the amount of trusted code is large. As a result, the simple server suffers from the higher relative cost of SFI instrumentation when compared to the Monkey HTTP server.

As described in Section 3.1, the cost of sandbox creation is independent of the size of the host process' page table. On creation of a NaCl sandbox, trusted code `mmap()`s the sandbox's address space to a zero-initialized copy-on-write shared-memory page (see `MAP_ANONYMOUS` in the `mmap(2)` man pages). A large NaCl module with significant internal state will therefore result in more traps to the kernel from protection faults than an equivalent NaCl with less code and less internal state.

In contrast, the creation of a process via fork requires the kernel to create a separate page table for the child and points each of its entries to the pages held by the parent process. The kernel then marks each of the pages as read-only, such that the first process to write to a given page will cause a protection fault [TB15, p. 742]. With copy-on-write, a parent process that has a large

amount of mutable state (i.e. the Monkey HTTP server) will result in a posteriori higher cost of forking than a similar process with much less mutable state (i.e. the simple web server).

To sum up, the combined short-term and long-term costs of forking in the simple web server is less than the equivalent cost for the Monkey HTTP server. Therefore, for the simple web server, creating a separate HTTP parser process per client request (shown in red on Figure 4.3) results in a higher throughput than creating a new NaCl sandbox per client request (shown in magenta on Figure 4.3). In contrast, the cost of creating a separate HTTP parser process per client request in Monkey (shown in red on Figure 4.5) is higher than creating a NaCl sandbox per client request (as shown in magenta on Figure 4.5).

Next, we analyze the end-to-end connection times for our simple web server. Figure 4.4 plots the medians (solid lines) and 99th percentiles (dashed lines) of the connection life times for the same test runs as the six implementations shown in Figure 4.3. The httperf man page defines the connection life time as "the time between a TCP connection is initiated and the time the connection is closed"[24]. Therefore, a successful connection includes the HTTP request / response round-trip time. For example, the blue square at an offered load of 500 client connections per second shows that in 99% of successful connections a connection lifetime lasted 5858 milliseconds or less. Likewise, for the blue cross, the same offered load shows that in 50% of successful connections a connection lifetime lasted 1153 milliseconds or less.

The median connection times for the baseline (solid green) simple web server, soft isolation with Unix processes (solid orange) and soft isolation with NaCl sandboxes are too close to zero to be distinguished from the horizontal axis.

The creation of a NaCl sandbox per client request for our non-optimized strong isolation variant (shown in solid blue) of the simple web server shows the highest connection times for successful connections. This is because the NaCl validator must first validate the untrusted .nexe (NaCl module executable) for each client request which adds a significant amount of latency to the overall connection time.

As the connection lifetime plotted in Figure 4.4 uses the same results as Figure 4.3, the outlier 99th percentile for the baseline simple web server (shown in dashed green) at an offered load of 4,750 connections/sec is the result of the same inaccuracies as the throughput outlier seen in Figure 4.3.

[24] *httperf man page*. httperf, http://www.hpl.hp.com/research/linux/httperf/httperf-man-0.9.txt, last accessed 17/08/2015

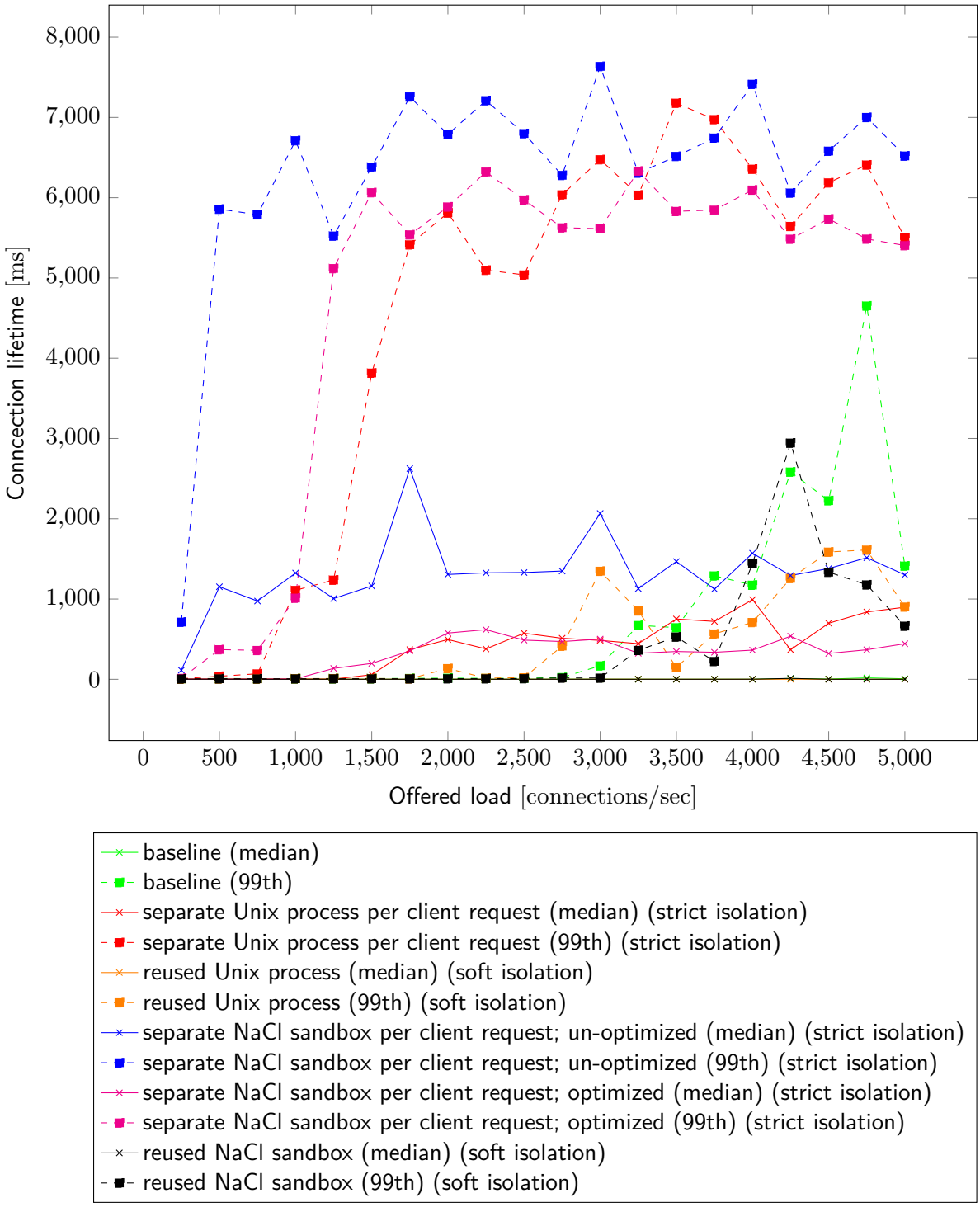

Figure 4.4: End-to-end connection life times for the simple web server

The next section conducts the same throughput and connection lifetime evaluations for the Monkey HTTP server.

4.2.3 Monkey HTTP server

We use the same experimental setup for the Monkey HTTP server as for the simple web server. However, we were unable to repeat each offered load for the strict isolation NaCl sandbox implementation because this particular version crashed after the creation of several thousands of sandboxes due to a NaCl `mmap` error.

Whilst there are a number of tutorials for debugging NaCl modules with NaCl's modified version of GDB, none of the team members were successful in figuring out how to use GDB without involving the Chrome browser. Questions in the Native-Client-Discuss on this topic have remained unanswered[25]. However, despite finding the exact line of source code where the `mmap` error occurred, we were unable to resolve it in the time that remained.

Figure 4.5 plots the throughput of our seven variants of the Monkey HTTP server. Recall that the soft isolation variants of the Monkey HTTP server create a separate reduced-privilege HTTP parser per Monkey worker thread and that Monkey creates one worker thread per virtual CPU core. However, since our evaluation ran on a single server with one virtual CPU core, these tests correspond to creating a single reduced-privilege HTTP parser which it reuses for all client requests.

Like the simple web server, the baseline variant of the Monkey HTTP server (shown in green) has the highest throughput of all server variants. The soft isolation implementations for Unix processes (shown in orange) and NaCl sandboxes (shown in black) result in near identical performance due to the similar cost of communication with Unix domain sockets and NaCl datagrams. This supports the results obtained in our microbenchmarks from Figure 4.2. The throughput for both variants is worse than baseline due to the overhead of copying message buffers between the trusted code and the reduced-privilege HTTP parser.

Like the simple web server, the non-optimized strict-isolation variant of the Monkey HTTP server has the worst performance. However, as explained prior, the optimized strict-isolation SFI variant which involves the creation of a NaCl sandbox per client request (shown in magenta) is faster

[25] *Maximum number of NaCl sandboxes within one process.* Google Groups Native-Client-Discuss, https://goo.gl/IbwRAx, last accessed 22/08/2015

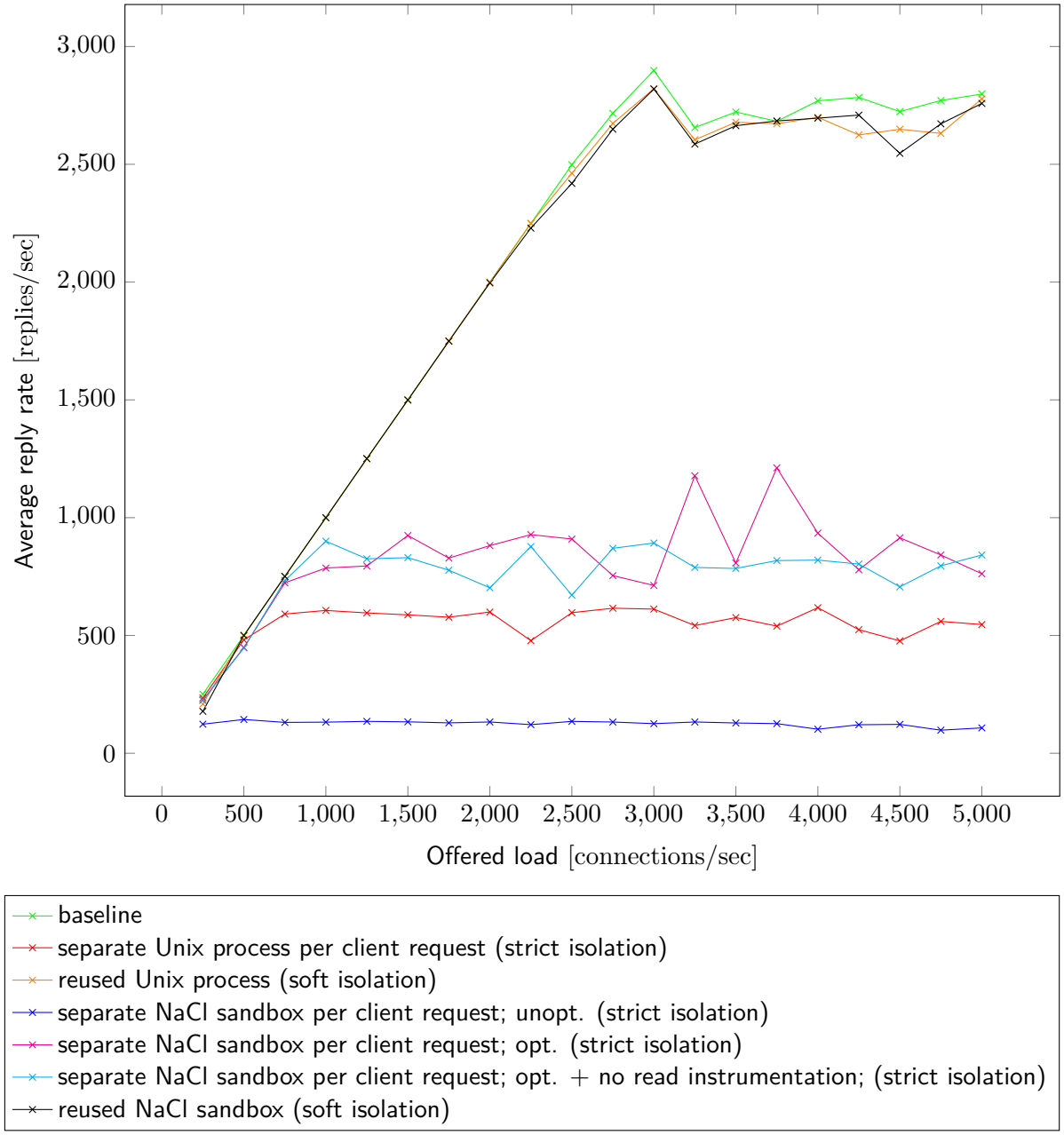

Figure 4.5: Throughput (replies/second) for the Monkey HTTP server

than the strict-isolation variant that uses Unix processes (which involves a `fork/exec` per client request) because forking is more expensive in the long-term for the Monkey HTTP server than it is for the simple web server. Figure 4.5 supports our overall hypothesis for this project that reduced-privilege compartments isolated via SFI have better performance than isolation via Unix processes.

The cyan line in Figure 4.5 represents the optimized strong-isolation SFI variant of the Monkey HTTP server that has been compiled with read masking disabled (see Section 3.8) such that it can read outside the sandbox. Figure 4.5 would indicate that the difference in performance from disabling read masking instructions is negligible.

It is worth highlighting that all server variants sustain a constant throughput having reached their respective peak rates. This sustained throughput is desired behavior as it affirms that no server variant suffers from receive livelock in the face of increasing load [MWMR97].

Like the connection lifetimes plotted for the simple web server, Figure 4.6 plots the connection lifetimes for the Monkey HTTP server.

The solid lines in Figure 4.6 depict the median connection life times whilst the dashed lines depict the 99th percentiles. Baseline Monkey (shown in green) exhibits the best connection life times followed by the soft isolation variant with Unix processes (shown in orange) and NaCl sandboxes (shown in black). The unoptimized strong-isolation NaCl sandbox variant (which creates a separate NaCl sandbox per client request) has the worst responsiveness.

However, the most important point of Figure 4.6 are the two red and magenta lines. We observe that the median connection lifetime for the strict-isolation server variant with Unix processes (in which the server forks per client request, shown in solid red) is 2.5x higher than the equivalent variant (strict-isolation with NaCl sandboxes) that creates a NaCl sandbox per client request (shown in solid magenta) across all offered loads. We contribute this increase in responsiveness to the fact that creating a NaCl sandbox is independent of the host process' memory footprint whereas the cost of forking is greater for "heavier" processes.

As with the throughput evaluation, the results in Figure 4.6 support our hypothesis that least privilege isolation via SFI has better performance than least privilege isolation via Unix processes for server side software.

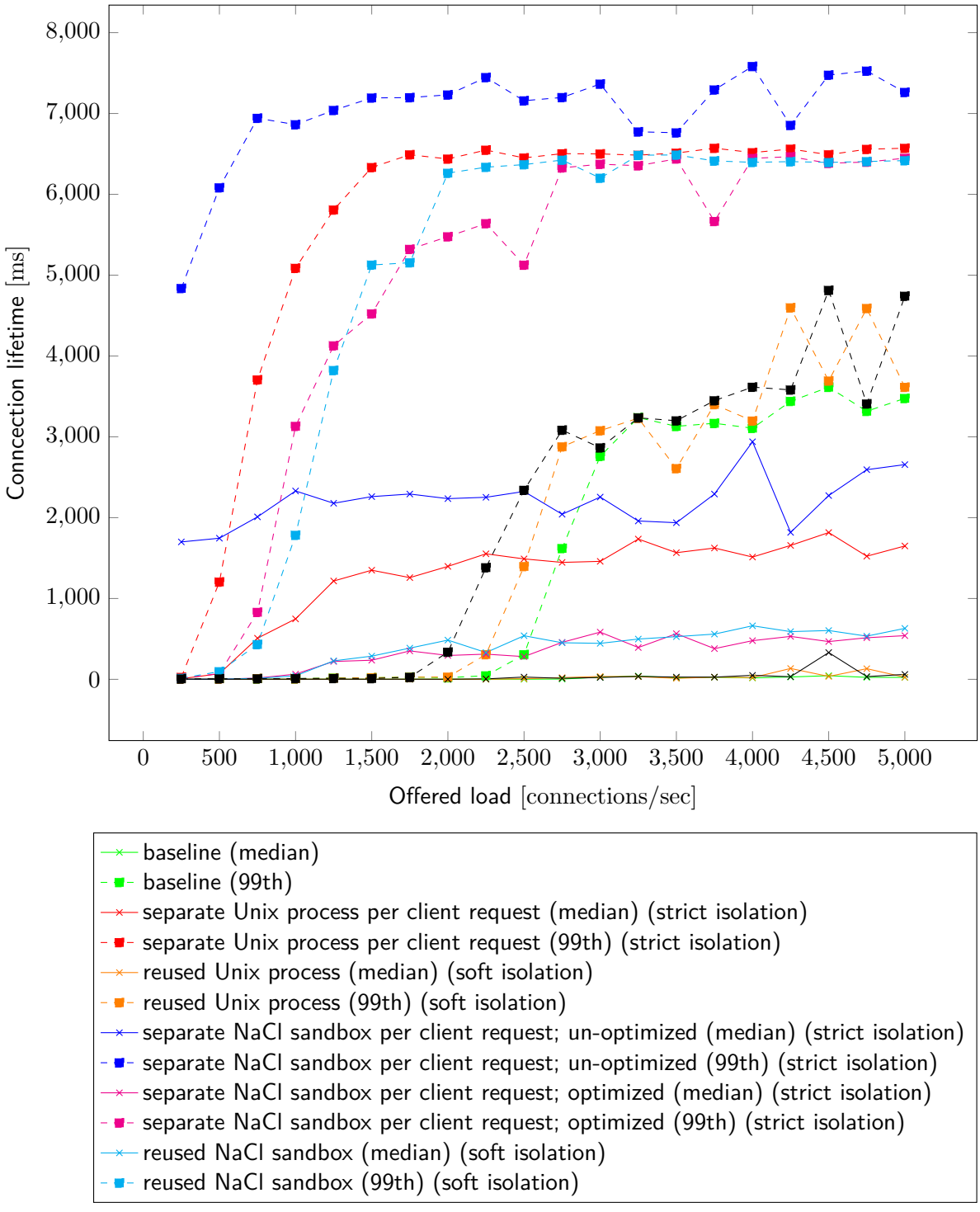

Figure 4.6: End-to-end connection life times for the Monkey HTTP server

4.3 Security evaluation

We now conduct our security evaluation of NaCl. First to demonstrate that untrusted code cannot read or write outside the sandbox we pass a pointer from trusted code to untrusted code via IMC and attempt to dereference the pointer in order to read and write at the address in trusted code. We then examine the sandboxing instructions which force the address stored in the pointer to point within the untrusted code's 4GB sub-address space.

Second, to demonstrate that untrusted code cannot execute trusted code we first exploit the HTTP parser of our baseline simple web server via a malicious HTTP request. The HTTP request overflows a buffer on the stack (see Section 2.3.3) in order to overwrite the return address and jump to a privileged function. We then demonstrate that the equivalent exploit for the sandboxed HTTP parser cannot jump to the same trusted code even though it resides within the same process address space.

4.3.1 Reading and writing outside the sandbox

To demonstrate that untrusted code cannot read or write outside the sandbox we first define a pointer to char in trusted code initialized with the string `printme`. We then pass the address of the pointer to the untrusted code via IMC, shown in Listing 4.3.

```
char *buf = "printme";
uint64_t ptr = (uint64_t) buf;
printf("ptr is %016llX\n", ptr);
printf("message is %s\n", (char *) ptr);
...
NaClIOVec iov = {ptr, sizeof(ptr)};
NaClMessageHeader msg = {&iov, 1, NULL, 0, 0};
NaClSendDatagram(IMC_DESC, &msg, 0);
```

Listing 4.3: Trusted code which defines a pointer to char and sends it to untrusted code over IMC.

The untrusted code then receives the pointer from trusted code over IMC and attempts to dereference it (shown in Listing 4.4). However, this dereference results in the crash of the entire process

(shown in Listing 4.5).

```
NaClRecieveDatagram(IMC_DESC, &msg, 0);
printf("received ptr is %016llX\n", msg.iov);
printf("message is %s\n", (char *) msg.iov);
```

Listing 4.4: Untrusted code which receives a pointer to char from trusted code over IMC.

```
$ ./trusted untrusted.nexe
[97086,3379595136:10:01:54.823583] Native Client module will be loaded at
    base ptr 0x0000146300000000
message is printme
char ptr is 00000000CA06E010
Segmentation fault (core dumped)
```

Listing 4.5: The untrusted code suffers a segmentation fault when it dereference the pointer.

To understand the cause of this segmentation fault we look at the object dump of the untrusted code (shown in Listing 4.6).

```
20280: 8b 9c 24 90 00 00 00 mov 0x90(%rsp),%ebx
20287: bf 1d 02 02 10 mov $0x1002021d,%edi
2028c: 31 c0 xor %eax,%eax
2028e: 89 de mov %ebx,%esi
20290: 66 66 2e 0f 1f 84 00 data32 nopw %cs:0x0(%rax,%rax,1)
20297: 00 00 00 00
2029b: e8 a0 24 00 00 callq 22740 <printf>
```

Listing 4.6: The object code for the untrusted code that dereference the pointer from trusted code as shown in Listing 4.4.

The first instruction in Listing 4.6 copies the address of the char pointer into the register %ebx and clears the upper 32 bits. The instruction at address 2028c clears the %eax register for use in the nop at address 20290. Last, the instruction at the address 2028e copies the address of the char pointer from %ebx to %esi for the argument to printf. The correct base address for the 4GB sub-address space is then added in the service runtime through which all system calls must

be mediated.

To demonstrate that untrusted code cannot write outside the sandbox we instead attempt an strcpy on the vector (shown in Listing 4.7).

```
strcpy((char *) msg.iov, "printed");
```

Listing 4.7: Untrusted code that attempts to read outside the sandbox.

However, as shown in Listing 4.8, this code too results in a segmentation fault.

```
$ ./trusted untrusted.nexe
[97697,2409158528:11:13:16.951672] Native Client module will be loaded at
    base address 0x0000732500000000
address is 00007F119049F010
message is printme
char ptr is 00007F119049F010
Segmentation fault (core dumped)
```

Listing 4.8: Output from untrusted code that attempts to read outside the sandbox.

Listing 4.9 shows the objdump of the untrusted code that attempts to read outside the sandbox. We can see that the instruction at address 202a0 copies the address in %ebx to %eax which clears the upper 32 bits. The compiler applies an additional masking instruction at address 202a2 and then adds the base address of the 4GB sub-address space stored in the designated register %RZP (%r15 in the current implementation of NaCl) to force the address within the untrusted code's fault domain.

```
00000000000201a0 <main>:
  ...
  202a0: 89 d8   mov %ebx,%eax
  202a2: 89 c0   mov %eax,%eax
  202a4: 41 c7 04 07 6e 65 77  movl $0x77656e,(%r15,%rax,1)
```

Listing 4.9: Objdump of untrusted code that attempts to read outside the sandbox.

Therefore, as shown in our demonstration above, each indirect load and store is replaced with sandboxing instructions which force the address within the untrusted code's fault domain. More-

over, we attempt to use hardcoded addresses to read and write from trusted code (with ASLR disabled). However, since untrusted code is compiled in ILP32 mode all pointers are 32 bits wide and therefore use 32-bit registers. As a result, a pointer cannot reference an address outside the sandbox.

4.3.2 Jumping to trusted code

In order to demonstrate that untrusted code cannot execute outside the sandbox we introduce an exploitable buffer overflow vulnerability in the HTTP parser of our simple web server. We craft a malicious HTTP request to overwrite the return address on the stack and jump to a privileged function for the baseline simple server. We then demonstrate that the equivalent exploit for the sandboxed HTTP parser cannot jump to the same trusted code even though it resides within the same process address space.

However, before we can exploit the HTTP parser via a malicious HTTP request we must first understand the data structures and control flow involved in the request-response cycle for an HTTP request.

Our simple web server serves clients in the following sequential steps:

1. The `simple_web_server_start` function sits in a tight loop accepting client connections (shown in Listing 4.10).
2. Upon return from `accept` it calls the `simple_web_server_do` function (shown in Listing 4.12).
3. The `simple_web_server_do` function calls `recv` on the file descriptor and blocks until the HTTP request is available to be read. Once available, the `recv` function reads the entire HTTP request into an 8KB buffer.
4. Upon return from `recv` the `simple_web_server_do` function invokes the `simplewebserver_http_parse_request` function in which it passes a pointer to the start of the buffer, the size of the buffer (in bytes) and a pointer to a stack allocated `simplewebserver_http_request_t` struct as the destination for the parsed HTTP request.
5. The `simplewebserver_http_parse_request` function tokenizes the HTTP request and fills in the `simplewebserver_http_request_t` struct one token at a time.

6. The `simplewebserver_http_parse_request` function returns control to the `simple_web_server_do` function which then completes the request-response cycle (i.e. generating the HTTP response headers and writing the requested resource out onto the network socket).

```
while(1) {
    addrlen = sizeof(clientaddr);
    clientfd = accept(server->sockfd, (struct sockaddr *) &clientaddr, &
        addrlen);
    simplewebserver_do(server, clientfd);
    close(clientfd);
}
```

Listing 4.10: The simple web server event loop.

```
    ...
    char buf[8192], filename[256];
    struct simplewebserver_http_request_t req;
    ...
    recv(clientfd, buf, sizeof(buf), 0);
    simplewebserver_http_parse_request(buf, sizeof(buf), &req);
    ...
```

Listing 4.11: The simple web server HTTP request handler.

The `simplewebserver_http_request_t` struct has just three fields: a short (two byte) int to represent the HTTP verb [FGM+99, 9. Method Definitions], a 2KB buffer to represent the resource [FGM+99, 5. Request] and a 32 byte buffer for the protocol, such as HTTP 1.0 and HTTP 1.1 (shown in Listing 4.12).

```
struct simplewebserver_http_request_header_t {
    unsigned short int verb;
    char resource[2084];
    char protocol[32];
};
```

Listing 4.12: Definition of the simplewebserver_http_request_header_t struct.

Exploiting baseline simple web server

In order to exploit our simple web server via a malicious HTTP request we make it possible for an attacker to overflow the 2KB resource buffer in the `simplewebserver_http_request_header_t` struct. To be precise, we make a buffer overflow possible via use of the unsafe `strcpy` string manipulation function (see line 2 of Listing 4.13) rather than the safe `strncpy` equivalent (see Section 2.3.1).

```
token = strtok(NULL, " ");
strcpy(req->header.resource, token);
```

Listing 4.13: Exploitable code that parses the resource of a HTTP request.

However, in order to overflow the resource buffer and overwrite the return address stored on the stack we need to know the distance between the start of the resource buffer in the `simplewebserver_http_request_t` struct and the return address (saved instruction pointer, `%rip`) stored on the stack. To calculate this distance we run the server under the GDB and inspect the stack. We take into account environment variables that might be pushed onto the stack prior to execution we mediate execution of the server via a bash script `r.sh`[26] which normalizes the different environment variables when run under GDB compared to normal program execution.

Figure 4.7 shows the stack layout of the `simplewebserver_http_request_do` function when compiled on GCC version 4.8.4 with the -fno-stack-protector option.

[26] https://github.com/hellman/fixenv/blob/master/r.sh

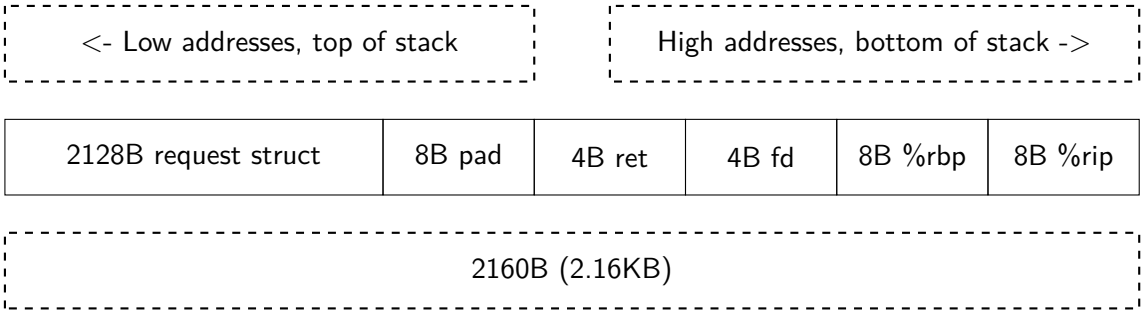

Figure 4.7: The stack layout of the simplewebserver_http_request_do function when run on x86-64 compiled with GCC version 4.8.4 and the -fno-stack-protector option.

We can see from Figure 4.7 that we must write 2150 bytes into the resource buffer in order to reach the start of the return address `%rip`. We know this because the resource buffer starts two bytes from the start of `struct simplewebserver_http_request_t` (due to the two byte short int for the HTTP verb) and the start of the `struct simplewebserver_http_request_t` struct is 2152 bytes from the return address `%rip`.

Given that we now know the distance between the start of the request struct and the return address we can craft a malicious HTTP request to overwrite the return address with another address of our choice. However, for the purpose of this evaluation we choose the address of a simple function that makes a single `printf` call as shown in Listing 4.14.

```
void
simplewebserver_dump_data()
{
    printf("%s\n", "Simple web server dump data");
}
```

Listing 4.14: The mock simplewebserver_dump_data function.

Listing 4.15 shows the object code for the dump data function. We can see from line 1 in Listing 4.15 that we must overwrite the return address for the `simplewebserver_http_parse_request` function stored on the stack with the address `00000000004013d0`.

```
00000000004013d0 <simplewebserver_dump_data>:
    4013d0: 55 push %rbp
    4013d1: 48 89 e5 mov %rsp,%rbp
    ...
```

Listing 4.15: The objdump of the simplewebserver_dump_data function in the baseline server.

Listing 4.16 shows our crafted exploit HTTP request. Its construction starts with a call to echo with the following options: -e to enable the interpretation of backslash escapes and -n to disable the output of the trailing newline. The elimination of the newline character is critical to our exploit since a newline character would write an additional byte `0x0A` (the newline character) into the return address. We then use perl to print the 'A' ASCII character 2150 times followed with the bytes `0xD0`, `0x13` and `0x40`. The final token is the protocol `HTTP/1.1`. The entire request string is then piped to telnet and sent to the vulnerable instance of our simple web server bound to the loopback address 127.0.0.1 and port number 4000.

Once a TCP connection has been established, the simple web server accepts the client connection and reads the malicious HTTP request into the buffer via the call to `recv`. It first parses the HTTP verb and translates it to an internal bit field representation. It then copies the first 2084 'A' characters into the resource buffer and writes the remaining 66 characters into the addresses below, overwriting the `ret`, `fd`, and saved base pointer on the stack (see Figure 4.7 for the stack layout). Last but not least, the bytes `0xd0`, `0x13` and `0x40` are written into the three least significant bytes of the return address such that we overwrite the saved instruction pointer with the address of the `simplewebserver_dump_data` function as shown in Listing 4.15.

```
echo -en "GET $(perl -e "print 'A'x2150")\xd0\x13\x40 HTTP/1.1" | telnet
    127.0.0.1 4000
```

Listing 4.16: A malicious HTTP request to exploit the HTTP parser of the simple web server and overwrite the return address with that of the simplewebserver_dump_data function.

Listing 4.16 shows the terminal output from the server. We can see that our malicious HTTP request overwrote the return address with the address of the `simplewebserver_dump_data` function whose output "*Simple web server dump data*" is printed on line 3.

```
~/simplewebserver$ ./r.sh ./simplewebserver -b 127.0.0.1 -p 4000
Listening on 127.0.0.1:4000.
Simple web server dump data
./r.sh: line 198: 27965 Segmentation fault (core dumped) env -i "${envlist[@
   ]}" "`pwd`/.launcher" "${@:2}"
```

Listing 4.17: Terminal output of the simple web server post exploitation.

4.3.3 Exploiting the sandboxed simple web server

Having exploited our baseline simple web server we now proceed to demonstrate that whilst we can still overwrite the return address of the sandboxed HTTP parser we cannot jump to an address outside the parser and into the trusted code. In order to show that the same exploit cannot execute trusted code from within the sandboxed HTTP parser we move the `simplewebserver_dump_data` function to the trusted code whilst preserving the `strcpy` vulnerability in the HTTP parser shown in Listing 4.13.

However, since we've split the code into two executables: one trusted executable for the server and one untrusted NaCl module for the HTTP parser, we must objdump the trusted code to find the address of the `simplewebserver_dump_data` function (shown in Listing 4.18).

```
00000000000070e0 <simplewebserver_dump_data>:
  70e0: bf 40 02 02 10 mov $0x10020240,%edi
  70e5: e9 b6 fc 01 00 jmpq 404e0 <puts>
```

Listing 4.18: Objdump of the simplewebserver_dump_data function in trusted code.

In addition, since NaCl uses ILP32 (x32 ABI) in untrusted code (for the 32-bit 4GB sub-address space) we must recalculate the offset from the request struct to the return address as a result of different structure padding and alignment rules. Without a functional GDB debugger for NaCl, we hexdump the stack of the sandboxed parser in order to find offset to the return address (shown in Figure 4.8).

Figure 4.8 highlights the 64-bit return address of the sandboxed HTTP parser. The first four bytes (shown in yellow) correspond to the four least significant bytes of the return address. The last four bytes (shown in red) correspond to the four most-significant bytes of the return address and represent the base address of the NaCl sandbox. However, even if we overwrite all eight bytes

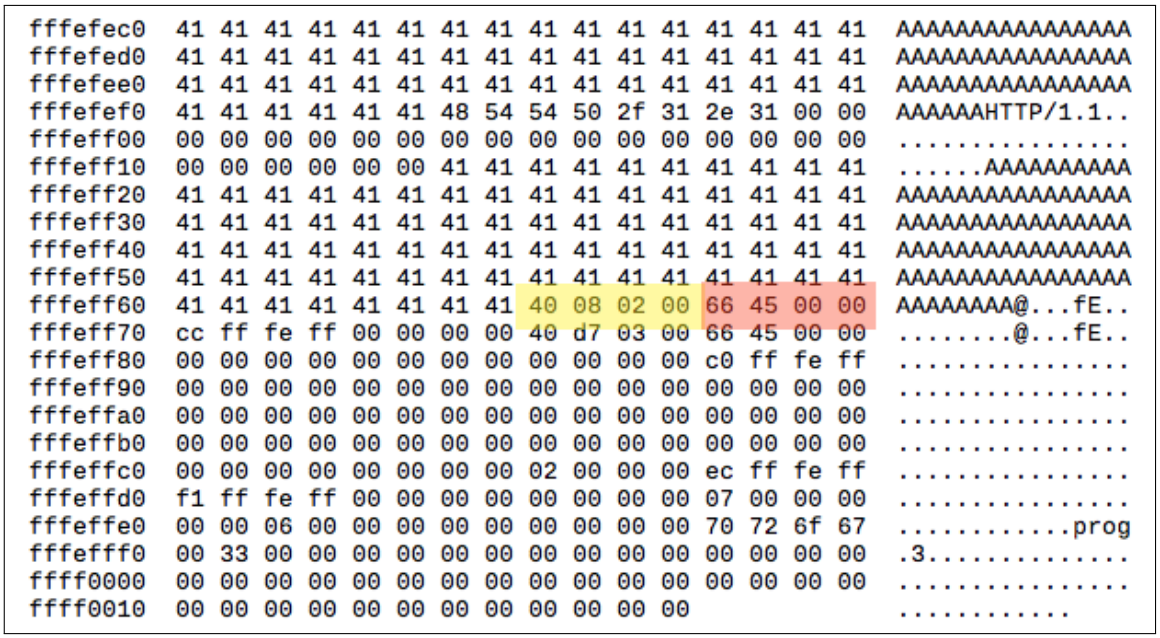

Figure 4.8: Hexdump of the stack in the sandboxed HTTP parser.

of the return address we hypothesize that we still cannot jump outside the sandbox due to the `nacljmp` pseudo-instruction shown in Listing 2.8.

With the correct offset from the resource buffer to the return address now identified we overwrite the return address of the HTTP parser with the address of `simplewebserver_dump_data` function in the trusted code via the malicious HTTP request shown in Listing 4.19.

```
echo -en "GET $(perl -e "print 'A'x2198")\xe0\x70\x00\x00\x00\x00\x00\x00
    HTTP/1.1" | telnet 127.0.0.1 4000
```

Listing 4.19: A malicious HTTP request to exploit the sandboxed HTTP parser of the simple web server and overwrite the return address with that of the simplewebserver_dump_data function in the trusted code.

However, on inspection of the overwritten return address we find that this exploit does not work. Whilst we can overwrite the two least significant bytes with `0xe0` and `0x70` we cannot overwrite the rest of the return address with `0x00`. This is because `0x00` is the NULL terminator character and will cause `strcpy` to return. Therefore, we end up with a return address with the higher order bits still in place.

Since our aim is to show that an exploit cannot break out of the sandbox rather than whether an attacker can write zeros on the stack with the `strcpy` function, we instead write a function called `overwrite_ret` that enables us to overwrite memory one byte at a time (shown in Listing

4.20).

```
void overwrite_ret(void *start, size_t offset, char *dst) {
    int i;
    char *ret = ((char *) start + offset);

    for (i = 0; i < 8; i++) {
        ret[i] = dst[i];
    }
}
```

Listing 4.20: The overwrite_ret function overwrites the return address *offset* bytes from the start address with the target address in dst.

We then call the `overwrite_ret` function in the HTTP parser as shown in 4.21.

```
overwrite_ret((void *) &req, 2200, "\xe0\x70\x00\x00\x00\x00\x00\x00");
```

Listing 4.21: The invocation of the overwrite_ret function in the HTTP parser to overwrite the return address with 0xe070.

Figure 4.9 shows the return address on the stack overwritten with the address of the dump data function in trusted code.

```
fffeff30  00 00 00 00 00 00 00 00 f1 ff fe ff 00 00 00 00  ................
fffeff40  d0 d6 fe ff 00 20 00 00 02 00 00 00 00 00 00 00  ..... ..........
fffeff50  02 00 00 00 00 00 00 00 cc ff fe ff 00 00 00 00  ................
fffeff60  d8 ff fe ff 00 00 00 00 e0 70 00 00 00 00 00 00  .........p......
fffeff70  cc ff fe ff 00 00 00 00 c0 d7 03 00 2e 44 00 00  .............D..
fffeff80  00 00 00 00 00 00 00 00 00 00 00 00 c0 ff fe ff  ................
```

Figure 4.9: The return address of the sandboxed HTTP parser overwritten with the address of the dump data function in trusted code.

Yet, when we run the server we cannot jump to the dump data function in trusted code. Instead, the HTTP parser crashes with a segmentation fault.

```
Sent 2124 bytes to trusted code
Segmentation fault (core dumped)
```

Listing 4.22: Output from exploited sandboxed HTTP parser.

This is because of the `nacljmp` pseudo-instruction at addresses 20530 and 20537 shown in Listing 4.23.

```
2052e: 41 5b pop %r11
20530: 41 83 e3 e0 and $0xfffffffe0,%r11d
20534: 4d 01 fb add %r15,%r11
20537: 41 ff e3 jmpq *%r11
```

Listing 4.23: Nacljmp instructions in objdump of the untrusted HTTP parser.

When the HTTP parser returns, the instruction at address 2052e pops the return address off the top of the stack and into the register %r11. However, since this is an indirect jump (recall that the ret instruction is disallowed in NaCl) the compiler must mask the offending instructions. The first of which, at address 20530, clears the top 32 bits (which correspond to the base address of the NaCl sandbox's 32-bit 4GB sub-address space) and enforces the 32 byte bundle alignment with the bitwise *and* operation on the literal value 0xffffffe0. The second instrumentation instruction at address 20534 then adds the base address for the NaCl sandbox's 32-bit 4GB sub-address space back in the top 32 bits of the address from the dedicated register %r15. With the address for the indirect jump now safe, the untrusted code is free to jump to the address.

4.4 Read sandboxing overhead

As described in Section 3.8, we attempt to remove the insertion of read sandboxing instructions in the modified compiler toolchains. However, we found that NaCl's isolation model requires the x32 ABI for x86-64 such that untrusted code is compiled in the ILP32 mode.

Yet, we found that changing the NaCl compiler toolchain to compile untrusted code in ILP64 mode would break the limitations enforced with displacement addressing (see Section 2.7.2) and thus enable untrusted code to both read and write outside of the sandbox. Furthermore, we have not been able to propose an alternative isolation scheme which would enable the safe execution of untrusted 64-bit compiled executables.

Nevertheless, we removed the addition of read masking instructions by changing both the GCC and LLVM compiler toolchains to prevent the insertion of masking instructions and the NaCl validator to ignore checks for the presence of such instructions.

To see if the overhead of masking instructions impacted the performance of a reduced-privilege web server we first look at the size of the compiled object code for the HTTP parser in the simple web server and Monkey HTTP both with and without the addition of masking instruction.

Web server	**Size with read masking**	**Size without read masking**	**Percentage decrease**
Simple web server	549.213 KB	548.965 KB	0.045%
Monkey HTTP server	662.005 KB	661.749 KB	0.039%

Table 4.1: Size of the object code (in KB) for the HTTP parser of both servers with and without read masking instructions

Table 4.1 shows the size of the object code for the HTTP parser in both server variants when compiled with and without masking instructions. We find that the addition of masking instructions has little effect on the size of the object code (a percentage decrease of 0.045% for the simple web server and 0.039% for the Monkey HTTP server).

The relationship between the size of the object code and the percentage decrease is non-intuitive. However, we identified that the padding to enforce 32 byte bundle alignment often filled up any gaps that resulted from the elimination of masking instructions. In addition, the masking instructions can be represented by fewer bytes than most other instructions. For example, the instruction `mov %eax, %eax` is two opcodes, `89 C0`, which is significantly shorter than most other instructions in x86-64.

Next, we use the callgrind analysis tool in Valgrind to compare the total number of instructions executed in the HTTP parser for the simple web server and Monkey HTTP server for both soft

(shown in Table 4.2) and strict isolation (shown in Table 4.3) with and without the addition of masking instructions.

Soft isolation				
Web server	**HTTP requests**	**Number of executed instructions with masking**	**Number of executed instructions without masking**	**Percentage decrease**
Simple web server	1	53,550	50,640	5.4342%
	2	104,921	99,135	5.5985%
Monkey HTTP server	1	38,738	37,901	2.1607%
	2	72,023	70,416	2.2312%

Table 4.2: The number of instructions executed in the HTTP parser for the simple web server and Monkey HTTP server for the soft isolation model.

Strict isolation				
Web server	**HTTP requests**	**Number of executed instructions with masking**	**Number of executed instructions without masking**	**Percentage decrease**
Simple web server	1	749,318	747,038	0.3042%
	2	1,383,196	1,378,182	0.4313%
Monkey HTTP server	1	616,025	614,882	0.1855%
	2	1,183,546	1,181,254	0.2024%

Table 4.3: The number of instructions executed in the HTTP parser for the simple web server and Monkey HTTP server for the strict isolation model.

Table 4.2 and 4.3 show our results. Whilst the percentage decrease in executed instructions is at best 0.43% for strict isolation, we find the percentage decrease to be as high as 5.59% for soft isolation.

First, we note that since x86 and x86-64 use the CISC architecture, instructions can be of variable length. Therefore, there is at best a weak correlation between the number of instructions and the program size. An object dump of both compiled HTTP parsers confirmed that parser modules without masking instructions have 6.8% fewer instructions overall (32,072 to 29,811).

Moreover, the percentage decrease for soft isolation is so much greater than strict isolation due to the implementation of each isolation method in our web server variants. For example, in soft isolation, a sandbox is created once at server initialization. This is in contrast to strict isolation in which a separate sandbox is created and destroyed per client request. Therefore, since Valgrind counts the total number of executed instructions for both trusted and untrusted code, we find that the overall difference is much smaller for the strict isolation method.

Last but not least, we consider whether the reduction in the size of the object code and number of executed instructions is of significance. If executing the masking instructions themselves is expensive we should see a significant performance increase for untrusted code. Therefore, to measure whether the cost of executing these masking instructions has a significant impact on web server performance we benchmarked the optimized, strict isolation SFI variant of the Monkey HTTP server without masking instructions to compare with our earlier results. However, as shown in Figure 4.5, the difference in performance that results from removing the addition of masking instructions is negligible.

4.5 Discussion

Whilst our evaluation is a strong representation of the performance of NaCl, we did not evaluate other SFI implementations. We stress that NaCl is an implementation of SFI to enable the safe execution of untrusted native code in the browser [YSD+09]. In a browser-based execution model, sandboxes are created and initialized once at web page load time. The frequent creation and destruction of sandboxes therefore falls outside the use-case that NaCl was intended for. Hence, there could be other implementations of SFI that outperform NaCl that we could not evaluate within the time constraints of the project.

However, as the results of our macrobenchmarks show, processes that have both a large memory footprint and a large amount of mutable state suffer from the cost of frequent page faults post fork due to copy-on-write. It should also be noted that the Monkey HTTP server's core code is small and modular when compared with other monolithic web servers such as the Apache HTTP server. For more complex web servers with higher memory footprints, such as the Apache HTTP server, we expect SFI to yield an even higher performance increase when compared to the same server whose reduced-privilege compartments are isolated via Unix processes.

Whilst our evaluation was further limited from the lack of a sufficient test bed, running all macrobenchmarks on a virtualized cloud service did ensure that our results were at least reproducible. Furthermore, in order to enable others to repeat or macrobenchmarks we publish the source code for all different server versions in addition to the raw results from httperf in our supporting documents. Our macrobenchmarks are neither reliable nor scientific due to the variance in performance on virtualized cloud services. As described in the experimental setup for our macrobenchmarks (see Section 4.2), we did not have control over the physical hardware, network conditions nor the

scheduling of other virtual servers on the same physical machine.

5 | Contributions

Our main contribution of this project is the demonstration that we can use SFI to build a reduced-privilege web server with both better performance and better security than the same server isolated via separate Unix processes. To the best of our knowledge, we built the first web server which enforces the Principle of Least Privilege via SFI.

Our first contribution towards SFI for reduced-privilege web servers is the separation of the HTTP parser for our simple web server into a separate reduced-privilege compartment sandboxed via SFI. Second, we demonstrated that untrusted code cannot compromise sensitive information via reading or writing static and global variables, the stack or heap of trusted code or via jumping to instructions in the trusted code's address space. This contribution serves as a practical demonstration to complement the proofs in [YSD+09, Figure 3]. Together, these two contributions stand to complete our first goal (see Section 1.4.1) derived from our high-level objective.

Third, we showed that SFI is a suitable mechanism for enforcing the Principle of Least Privilege in large and complex software by sandboxing the HTTP parser for the Monkey HTTP server. Furthermore, our results indicate that our SFI sandboxed Monkey server has better performance than the same server sandboxed via Unix processes (see Section 4.2). This contribution stands to complete our second goal (see Section 1.4.1) from our high-level objective.

Fourth, we made significant contributions to the implementation of NaCl. In order to support sandbox destruction we implemented a sandbox destructor function `NaClAppDtor` in the NaCl source code. We disabled the addition of read masking instructions in the NaCl compiler toolchains and modified the validator to enable the execution of untrusted code in the absence of these instructions. We measured the performance of the optimized, strict isolation Monkey HTTP server with and without read masking and compared the number of instructions in the compiled object code between the two. This contribution stands to complete our third and final critical goal (see Section 1.4.1) through which we realize our high-level objective.

To sandbox the SSL module we placed the entire Monkey HTTP server excluding its SSL module in untrusted code. This required modifications to the NaCl source code in order to support disallowed system calls such as creating, listening and binding to sockets. However, we were unable to complete our fourth and single non-critical goal (see Section 1.4.1) in the time that remained.

6 | Future Work

In this chapter we identify a number of areas that we would like to explore in future work. The areas that we present are the result of extensive discussions with our project supervisor, studying the source code and implementation of NaCl, and our analysis of the evaluation results.

6.1 Sandbox creation

The significant contributions of our work show that we can use SFI to build reduced-privilege web servers with better performance than the same web server isolated via Unix processes. However, this increase in performance comes at the cost of disabling validation of untrusted code during sandbox creation. Therefore, whilst we chose to trust the compiler for the purpose of our project (see Section 1.3), in practice compilers are complex software and like all complex software, compilers can and do contain faults.

We therefore propose the following improvement to the NaCl implementation which preserves the performance increase seen for non-verified untrusted code whilst still ensuring that untrusted code has the correct sandboxing instructions. We refer to this improvement as *hash-based validation*, described below.

One limitation of the current NaCl implementation is that it requires repeated validation of the same untrusted code for each subsequent sandbox creation. In the case of our strong isolation reduced-privilege web server, NaCl revalidates the same untrusted code for each incoming HTTP request even though the untrusted code itself is unchanged.

In hash-based validation the trusted code (within which the validator runs) asserts that the untrusted code contains the correct sandboxing instructions. If the untrusted code passes validation then the trusted code takes the hash of the untrusted code and stores it in trusted memory. When

the trusted code comes to create subsequent sandboxes for the same untrusted code it can first check whether the two hashes match. If so, the trusted code can execute the untrusted code without prior revalidation. If not, the validator must reassert that the untrusted code contains the necessary sandboxing instructions prior to updating the hash and initiating execution. As with our existing threat model, this modification assumes that trusted code cannot be exploited and that the hash function is collision, pre-image and second pre-image resistant.

However, reading in and hashing the untrusted code from disk is slow. Therefore, as an optimization we propose *one-time validation*. Like hash-based validation, one-time validation requires that untrusted code is verified on first run. However, once verified the untrusted code is cached in trusted memory. When the trusted code comes to create subsequent sandboxes from the same untrusted code it loads the untrusted code from the cache and never from the executable stored on disk. This method avoids the need for expensive hash computations at the cost of increased memory usage.

Another area for potential future work involves reducing the number of system calls involved in sandbox creation. Whilst performing our microbenchmarks of NaCl we decided to run a system call trace with the `strace` tool on sandbox creation to see how the 4GB address space and 40GB guard zones were mapped. To our surprise we found that sandbox creation incurred over 60 system calls compared to the single system call required for `fork()`. We hypothesize that the excessive number of mode switches to facilitate these system calls has a significant reduction on the performance of sandbox creation. We therefore propose an investigation into whether the number of system calls can be reduced for sandbox creation.

In addition to the large number of system calls that result from sandbox creation, NaCl reserves 84GB of the 64 bit virtual address space for each created sandbox. Whilst the two 40GB guard zones consume a large amount of the virtual address space, their cost of initialization is small since all their pages are unmapped. Therefore, the Linux kernel can set the page protection as unmapped for the respective third level page table. In contrast, the 4GB address space requires different protections for the service runtime, the program text and the region for the untrusted code's static and global variables, heap and stack. As a result of the four level page table in x86-64[27], this requires a significant number of modifications to fourth level page table entries. We therefore propose an investigation into whether this cost is of significance and whether reducing

[27] *AMD64 Architecture Programmer's Manual*. AMD, http://developer.amd.com/wordpress/media/2012/10/24593_APM_v2.pdf#180, last accessed 21/08/2015

the size of the 4GB address space might further increase the performance for sandbox creation.

6.2 Sandbox destruction

Whilst our implementation of sandbox destruction in NaCl releases almost all resources acquired in sandbox creation, there are a number of leaks that have yet to be fixed. Overall, the amount of leakage is small: just 2KB per sandbox. However, it results in eventual termination of the process by the host operating system after sufficient number (approximately one million) of sandboxes have been created and destroyed.

6.3 Checkpoint and restore

Rather than incur the cost of sandbox creation and destruction for each HTTP request we propose an implementation of the `checkpoint` and `restore` functions from Wedge [Bit09, 3.9.3] to enable safe sandbox recycling. The `checkpoint` and `restore` functions preserve the strong isolation guarantees of creating a separate sandbox per client request by creating a pristine snapshot of the sandbox prior to execution. Then, once the sandbox has terminated, a worker thread can restore its state via the `restore` function. The `restore` function unmaps the pages not present at the time of `checkpoint` and copies the checkpointed copy-on-write pages back into the sandbox [Bit09, p. 54]. Once restored, the recycled sandbox can be returned to the pool of available sandboxes. Last, we propose an evaluation to compare the performance of sandbox creation and destruction (both with and without our optimizations from Section 6.1) with the `checkpoint` and `restore` functions.

6.4 Inter-module communication

Our microbenchmarks in Section 4.1.2 compare the performance between IMC and IPC. Our results show that NaCl datagrams have near the same or worse performance than Unix domain sockets despite the sender and receiver being within the same address space. We therefore propose an investigation into the implementation of NaCl datagrams to see whether messages are mediated via system calls to the kernel (since IMC also supports IPC). If so, we propose that

the IMC implementation is modified to use shared memory for communication within the same address space whilst preserving the original implementation for communication between processes.

6.5 Monkey HTTP server

We now finish with future work for our SFI implementation of the Monkey HTTP server.

First, whilst we isolated the HTTP parser of Monkey into an SFI sandboxed reduced-privilege compartment, we found that the server would crash after a sufficient number of sandbox invocations. In particular, `mmap()` would fail to reserve the 84GB of the virtual address space required for the sandbox. We therefore propose an investigation to establish the cause of this error despite having `ulimit -v` set to unlimited and sufficient physical memory available.

Second, we were unable to complete the sandboxing of the SSL module for the Monkey HTTP server in the time that remained. We therefore propose the continuation of this work in addition to the separation of other modules into their own reduced-privilege compartments.

7 | Project Management

In this chapter we describe the project management techniques elected for our project. We describe our stakeholders who we illustrate with a detailed stakeholder map (Figure 7.1). We present out project schedule and describe our team organization which includes our work plan, communication platforms and our software development methodology. Last, we present our risk management strategy in which we rank the risks of our project and outline policies that we developed to help mitigate them.

7.1 Stakeholders

As this is a research project without industrial involvement, our real stakeholders are limited to the individual team members and our project supervisor. However, we present the stakeholders who would have been involved had we undertaken this project in an industrial setting.

At the core of our stakeholder map (Figure 7.1) is our intended product: our reduced-privilege web server.

The core team members, which consists of the students enrolled in the project, span all four layers of the stakeholder map as all members of the team are responsible for communication with the other stakeholders.

The operational work area includes normal operators, such as the end users that interact with our reduced-privilege web server via their web browser, and the operational support operators which represent the individuals and organizations managing deployments of the server.

The containing business includes the clients and sponsors who are paying for the product's development. In our project the clients and sponsors represent the core team members since each team member has invested their own time and tuition fees in the product's development. The

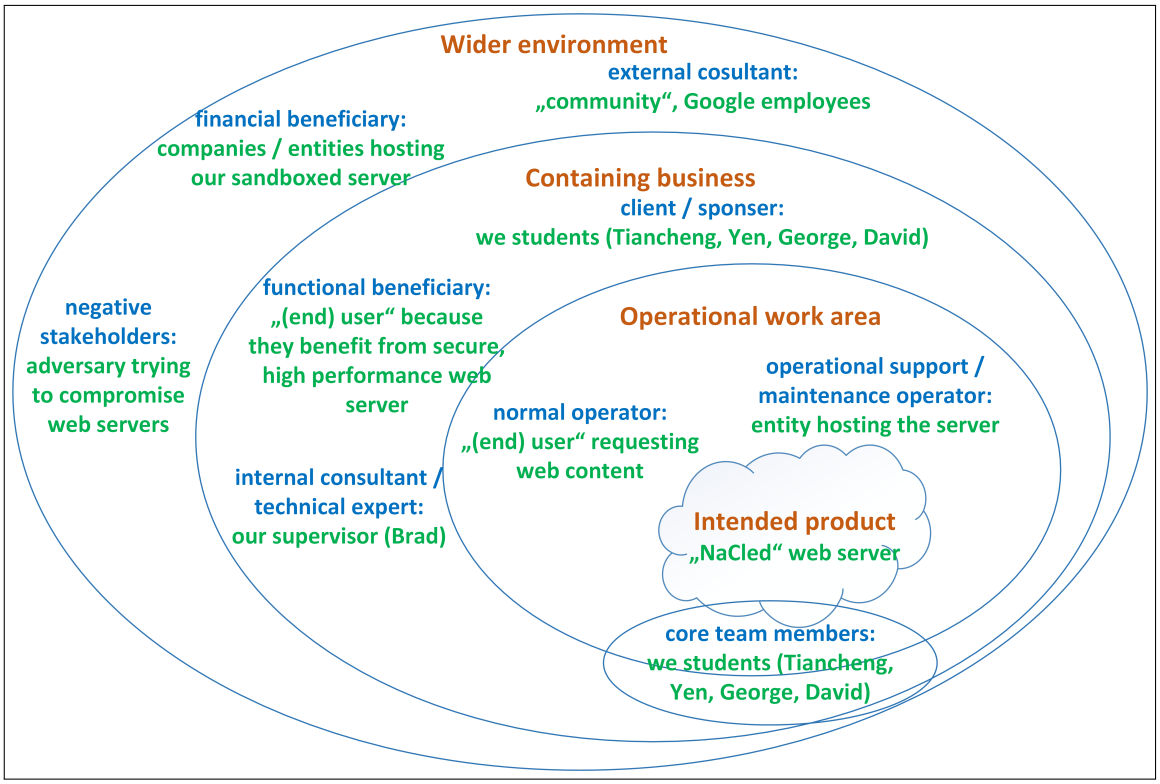

Figure 7.1: Stakeholder map

internal consultant is our supervisor. He offers technical expertise, answers student questions and helps guiding and directing our project. End users which interact with our reduced-privilege server assume the role of the functional beneficiary since they benefit implicitly from a secure, reduced-privilege web server.

The wider environment includes external consultants. The external consultants consist of the Google employees and open-source contributors which answer our technical questions on public discussion forums such as the official "Native-Client-Discuss"[28] Google group.

Our financial beneficiaries consist of the individuals and organizations that run our reduced-privilege web server. Their benefits include a limitation of financial and reputational damage. Data theft, loss or corruption could otherwise result from running a vulnerable, non reduced-privilege web server.

Last but not least, our negative stakeholders include the adversaries which seek to exploit our reduced-privilege web server.

[28] *Native-Client-Discuss*. Google, https://groups.google.com/forum/?fromgroups#!forum/native-client-discuss, last accessed 11/08/2015

7.2 Project schedule

Figure 7.2 shows our project schedule. We subdivide each high-level task (shown in green) into smaller and more manageable subtasks (shown in orange). Time is depicted on the horizontal axis whilst tasks are shown on the vertical axis.

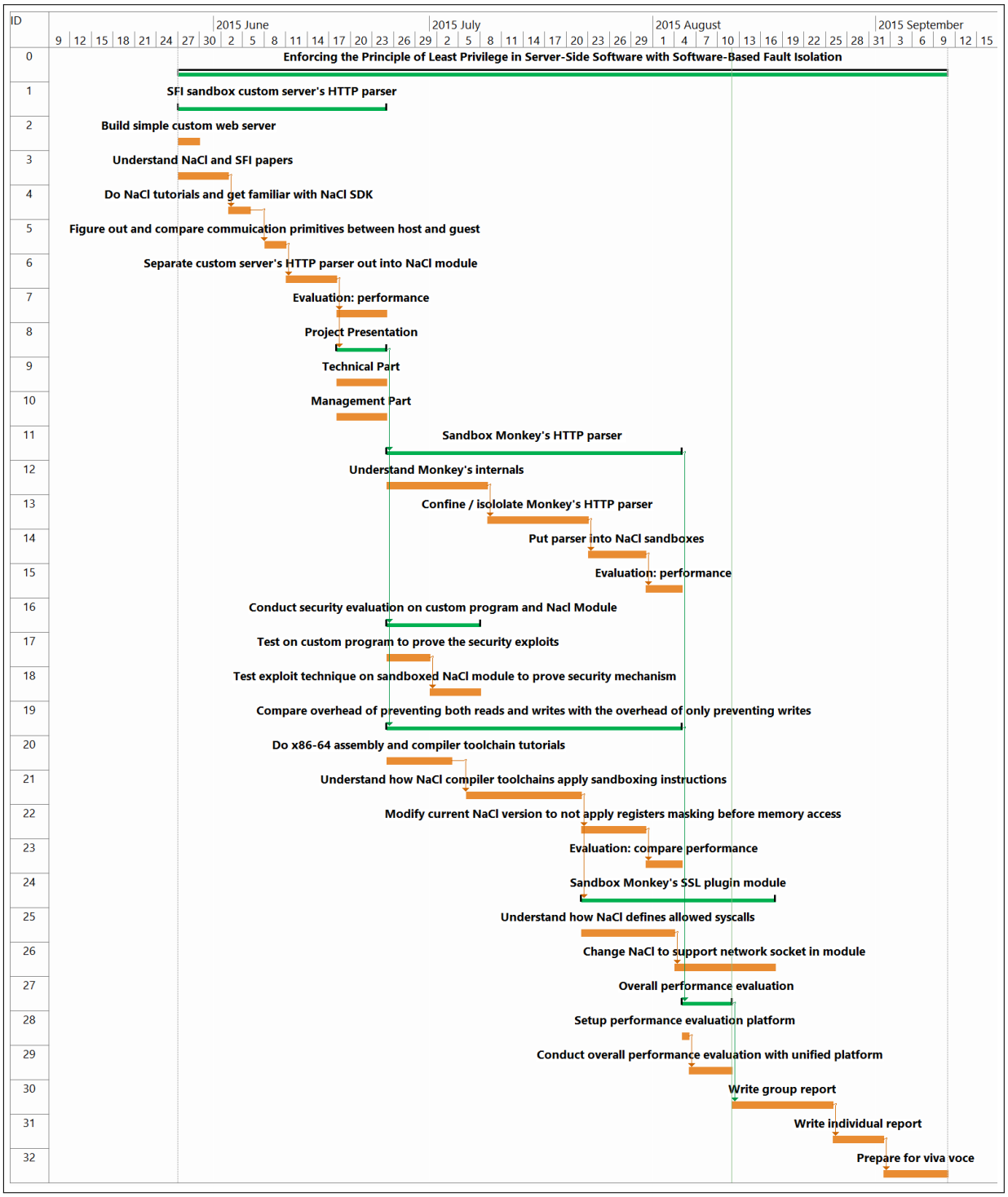

Figure 7.2: Project schedule

The high-level tasks (shown in green) correspond each of our goals from Section 1.4. Figure 7.2 shows our first milestone in which we sandbox the HTTP parser of our simple web server. Once our first milestone has been reached we then move on to sandbox the HTTP parser for the Monkey HTTP server: our HTTP standards compliant and feature complete HTTP server.

We start with the simple web server in order to familiarize ourselves with the NaCl internals and build the additional features required in order to support sandbox destruction (see Section 3). This has the added benefit of minimizing our risks and reducing the cognitive overhead of having to work with a complex HTTP server in tandem.

We parallelize the tasks of sandboxing the HTTP parser for the Monkey HTTP server with the performance evaluation for read sandboxing. In order to parallelize these tasks we divide our project group into two sub teams to whom we delegate one task each. We plan sufficient time in order to write the group and individual reports. We keep all provisional results documented and up to date in live web documents on Basecamp. Our workplan in Section 7.3.1 explains the delegation of tasks in more detail.

7.3 Team organization

In this section we explain how work was split up amongst team members. We describe our communication platforms and discuss how decisions were made and consensus was reached. Last we outline our project management approach and justify our choice of software development methodology.

7.3.1 Work plan

The project group consists of four team members: David Ansari, Yen-Pin Huang, George Robinson, and Tiancheng Yi.

We elected George for the project manager role. As project manager, George is responsible for the mediation of communication between the project group and our supervisor. In addition, his responsibilities include managing the progression of tasks in our project schedule (Figure 7.2) and organizing the weekly agendas for our supervisor.

Since this is a research project, we assigned all team members with the role of developer since everyone must contribute to the technical realization of our high-level objective.

Collaborative work was planned during *sprint planning meetings*. With a quorum present, we ranked the high-level tasks derived from our objective by their respective priorities in the form of a document known as a sprint backlog. The delegation of tasks amongst individual team members was decided together as group rather than being issued from the project manager. This meant that each individual team member could pick and choose the tasks that appealed to them most which improved motivation and job satisfaction across the entire team.

We now outline the delegation of tasks and responsibilities of each team member as decided in our aforementioned *sprint planning meetings*. Note that the contributions of individual team members are reserved for the individual reports.

Given that our first milestone is a prerequisite for all remaining tasks we coordinated the entire project team to work on sandboxing the HTTP parser of our simple web server. In addition to satisfying our prerequisite, this collective coordination enabled the entire group to become familiar with the fundamentals and implementation of NaCl in a short amount of time through a mechanism known as divide and conquer. In order to share knowledge of NaCl's implementation each team member contributed to a series of short documents to describe the behavior of NaCl's internal functions and the flow of execution for creation and sandbox initialization within NaCl.

With the first milestone complete we then subdivided tasks amongst individual team members. Whilst David started to sandbox the HTTP parser for the Monkey HTTP server, George began his security evaluation for the vanilla and sandboxed versions of the simple web server. Yen and Tiancheng worked together in order to analyze the performance of sandboxed read instructions.

Having completed the security evaluation, George then joined David in sandboxing the HTTP parser for the Monkey HTTP server. In mid-July Tiancheng moved onto sandboxing the SSL module of the Monkey HTTP server which he continued working on until the end of the project. With the sandboxing for the Monkey HTTP parser completed David, George and Yen started the macrobenchmarks for the different server implementations (section 4.2).

7.3.2 Team communication

We chose Basecamp[29] as our project management and communication platform since it supports group discussions, to-do lists, events, deadline management and file sharing. It serves as our hub for essential project information. Team members received notifications of activity via email and

[29] *Basecamp*. https://basecamp.com/, last accessed 12/08/2015

the smartphone app.

We chose Git as our revision control system since it fulfills the requirements of our distributed, non-linear workflows. To enable merging individual contributions into a single linear timeline we set up a private Git server on which we hosted repositories for our code and group report. In addition to communicating via Basecamp, our team meets Mondays to Fridays in the UCL Computer Science lab 4.06. We document all decisions on Basecamp.

7.3.3 Software development methodology

For our project we adopted the *agile* software development methodology. The main reason for going *agile* is a strong reduction of risks (see section 7.4). Through the frequent delivery of working software, starting with our first sprint, we decrease the likelihood of complete project failure. This is known in agile circles as *continuous delivery*.

In addition to reducing risks, agile welcomes changing requirements that may result from new information otherwise unavailable at the start of the project as well as frequently changing, volatile environments. One example of where we have benefited from changing requirements includes the outdated information in the NaCl papers. Therein, the authors describe the Simple Remote Procedure Call and Netscape Plugin API communication modules which are now both deprecated. Another example of where we have benefited from changing requirements includes the addition of read sandboxing in latest versions of NaCl, not mentioned in the original works [YSD+09] [SMB+10]. This discovery necessitated reevaluating one of our original goals in which we had intended to implement read sandboxing ourselves. As a result of welcoming changing requirements we decided to measure the performance overhead of read sandboxing in their current implementation. Finally, prioritization of the sprint backlog helped ensure that we achieved our critical goals before considering our non-critical goals which are non-critical to our high-level objective.

To help stay on track, each Wednesday afternoon the team meets with our project supervisor, Prof. Brad Karp. A weekly agenda is prepared and emailed 24 hours in advance. The agenda describes the work done since the last meeting from the week prior, the work planned for the current week and a list of questions for our supervisor. This provides our supervisor with a sufficient window of time in which to prepare answers to our questions and any topics for further discussion. We include our weekly agendas in the supporting documents submitted in conjunction

with this report.

One final aspect of our agile management approach is pair programming. Often, two team members worked together on the same machine. Whilst one wrote code, the other reviewed each line as it was written. This helped spot errors early which was critical given long build times for NaCl modules. In addition, the peer review that resulted for pair programming helped improve code quality. Furthermore, pair programming enabled the constant transfer of knowledge between team members and provided redundancy should one team member become unavailable or fall ill.

7.4 Risk management

Table 7.1 shows the risks to our project and the strategies we developed to help mitigate them. We categorize each risk with a low, medium or high probability. This probability quantifier is our estimated likelihood for each risk. In addition, we highlight the effect on our project should the event occur and describe our mitigation strategy in order to reduce its impact.

Type of Risk	Description	Probability	Impact	Mitigation strategy
Technical	Applying NaCl's design to server side applications it was originally not intended for	Medium	Medium	Change NaCl's internal implementation; consult Google development group
Design	Choosing a wrong design	Low	High	Weekly consultancy with supervisor
Change management overload	Increase in complexity; modification of original goals due to changed requirements	Medium	Low	Change management is a key component in our agile management approach
Resources and capabilities	Acquiring new skills may lead to delay of building the actual product	Medium	Medium	Split work among team members who become specialists on their assigned tasks
Communication	Group members might have different points of view	Medium	Low	Daily face-to-face meetings and shared platform for continuous interaction

Table 7.1: Project risks

8 | Conclusion

To review, in this Master's project we made significant contributions to the state of the two most fundamental characteristics we expect from contemporary web servers: speed and security. Since vulnerabilities in today's web servers are diverse and numerous, rather than attempt to prevent all vulnerabilities that may be present in software, we instead limit the extent of damage that could result from the successful exploitation of software vulnerabilities. In order to achieve this we split the functional components of a web server into isolated compartments consistent with a well known security principle: the the Principle of Least Privilege [SS75].

State of the art research applies the Principle of Least Privilege by isolating the functional components of a web server into reduced-privilege compartments that run inside separate Unix processes [BMHK08] [Kro04]. This project's focal problem statement claims that creation of processes and communication across processes is too expensive to preserve the high performance we expect from modern web servers. Rather than isolating reduced-privilege server compartments via separate processes, we took an alternative method that was Software-Based Fault Isolation [WLAG93]. Despite the fact that SFI itself was published over 20 years ago, to the best of our knowledge it has not yet been applied to the isolation of compartments in a reduced-privilege web server.

Chapter 2 of this report provided background knowledge of the Unix process model, the memory layout of `C` programs, buffer overflow exploits and the Principle of Least Privilege. We then described the underlying concepts of SFI before concentrating on the specific SFI implementation within Google Native Client. In Chapter 3 we explained our process and the technical decisions involved in splitting the HTTP parser of the simple and Monkey HTTP servers into a separate reduced-privilege compartment.

Our evaluation results presented in Chapter 4 support our hypothesis that a reduced-privilege web server isolated via SFI can achieve both better performance and stronger security than isolation via Unix processes. However, our results in Section 4.4 show that the performance improvement

from enabling a compartment to read outside the sandbox is negligible.

Therefore, this project's key contributions are as follows:

1. The implementation of a concurrent reduced-privilege web server isolated via SFI. To the best of our knowledge, this is the first reduced-privilege web server isolated via SFI
2. An evaluation which indicates that SFI can provide both better performance and stronger security than the same reduced-privilege web server isolated via Unix processes as implemented in cutting-edge research [BMHK08] [Kro04]. Our results show that isolation via SFI yields an average increase in sustainable load of 33% (see Figure 4.5) and a decrease in median latency of 60% (see Figure 4.6).

The main challenge we faced in this project was the application of NaCl's implementation of SFI, itself built for the safe execution of untrusted code in the browser, to server-side software. We faced non-trivial obstacles such as supporting the continuous creation and destruction of sandboxes within the same long-lived host Unix process and the secure communication between trusted code in the host process and untrusted code within the NaCl sandbox.

Chapter 6 identified a number of optimizations we would like to realize for future work. We conjecture that an SFI implementation optimized for server-side software could result in an even higher performance increase than our modified implementation of NaCl.

In conclusion, we realized all our critical goals each making valuable contributions to the wider research area of web server security.

Bibliography

[Bit09] Andrea Bittau. Toward least-privilege isolation for software, 2009.

[BMHK08] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into reduced-privilege compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 309–322, Berkeley, CA, USA, 2008. USENIX Association.

[CKNZ11] Edmund M. Clarke, William Klieber, Milos Novácek, and Paolo Zuliani. Model checking and the state explosion problem. In *Tools for Practical Software Verification, LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, pages 1–30, 2011.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Rfc 2616, hypertext transfer protocol – http/1.1, 1999.

[FKK11] A. Freier, P. Karlton, and P. Kocher. The Secure Sockets Layer (SSL) Protocol Version 3.0. RFC 6101 (Historic), August 2011.

[HYH+04] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 40–52, New York, NY, USA, 2004. ACM.

[Kro04] Maxwell N. Krohn. Building secure high-performance web services with okws. In *USENIX Annual Technical Conference, General Track*, pages 185–198. USENIX, 2004.

[MWMR97] Jeffrey Mogul, Dec Western, Jeffrey C. Mogul, and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer*

Systems, 15:217–252, 1997.

[One96] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996.

[SMB+10] David Sehr, Robert Muth, Cliff L. Biffle, Victor Khimenko, Egor Pasko, Bennet Yee, Karl Schimpf, and Brad Chen. Adapting software fault isolation to contemporary cpu architectures. In *19th USENIX Security Symposium*, pages 1–11, 2010.

[SPP+04] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization, 2004.

[SS75] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE 63-9*, 1975.

[TB15] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. Pearson, 2015.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *SIGOPS Oper. Syst. Rev.*, 27(5):203–216, December 1993.

[You13] Yves Younan. 25 years of vulnerabilities: 1988-2012. 2013.

[YSD+09] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Orm, Shiki Okasaka, Neha Narula, Nicholas Fullagar, and Google Inc. Native client: A sandbox for portable, untrusted x86 native code. In *In Proceedings of the 2007 IEEE Symposium on Security and Privacy*, 2009.

9 | Appendix

```
Accepted connection from 10.131.12.115, port 44867
[ 5] local 10.131.8.80 port 2001 connected to 10.131.12.115 port 44868
[ ID] Interval Transfer Bandwidth
[ 5] 0.00-1.00 sec 107 MBytes 895 Mbits/sec
[ 5] 1.00-2.00 sec 112 MBytes 938 Mbits/sec
[ 5] 2.00-3.00 sec 112 MBytes 938 Mbits/sec
[ 5] 3.00-4.00 sec 112 MBytes 938 Mbits/sec
[ 5] 4.00-5.00 sec 112 MBytes 939 Mbits/sec
[ 5] 5.00-6.00 sec 112 MBytes 939 Mbits/sec
[ 5] 6.00-7.00 sec 112 MBytes 939 Mbits/sec
[ 5] 7.00-8.00 sec 112 MBytes 939 Mbits/sec
[ 5] 8.00-9.00 sec 112 MBytes 939 Mbits/sec
[ 5] 9.00-10.00 sec 112 MBytes 939 Mbits/sec
[ 5] 10.00-10.09 sec 9.97 MBytes 937 Mbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval Transfer Bandwidth Retr
[ 5] 0.00-10.09 sec 1.10 GBytes 937 Mbits/sec 0 sender
[ 5] 0.00-10.09 sec 1.10 GBytes 934 Mbits/sec receiver
```

Listing 9.1: The output from the iperf tool to measure the network bandwidth available between the server and load generators on the internal LAN.

```
import argparse
import os
import subprocess
import sys
```

```
import time

'''
This benchmark runner starts httperf on our load generators then writes the
    results from stdout to the destination log file.

The benchmark runner requires the number of clients, the address and port
    number of the server to benchmark, the benchmark id (a string or integer
    description for the benchmark), the start rate (in req/s) and the max rate
     (in req/s). It then benchmarks the server repeat times at the start rate,
     increasing in steps of step-rate until the max rate.

For each of the client (load generators) the benchmark runner creates a
    seperate sub process in which it runs the httperf program via SSH. It then
     pipes stdout back into the process and writes it to the file.
'''

# The base hostname of the client load generator.
clienthost = "perf@ncs-group-project-load-gen{}.grobinson.net"

# The command to execute on the client.
cmd = "export PATH=$PATH:~/httperf-0.9.0/src; httperf --hog --server {server}
     --port {port} --rate {rate} --num-conn {conn} --client={client}/{clients}
     --timeout=5 -v --verbose"

parser = argparse.ArgumentParser(description='Benchmark runner')

parser.add_argument('clients',
                    metavar='clients',
                    type=int,
                    help='The number of clients')

parser.add_argument('addr',
```

```
                  help='The IP address or hostname of the server to benchmark
                      ')

parser.add_argument('port',
                  type=int,
                  help='The port number of the server to benchmark')

parser.add_argument('bid',
                  metavar='benchmark-id',
                  help='The benchmark id')

parser.add_argument('start_rate',
                  metavar='start-rate',
                  type=int,
                  help='The start rate in req/s per client')

parser.add_argument('max_rate',
                  metavar='max-rate',
                  type=int,
                  help='The maximum rate in req/s per client')

parser.add_argument('--step-rate',
                  default=25,
                  type=int,
                  help='The step rate in req/s per client')

parser.add_argument('--repeat',
                  default=1,
                  type=int,
                  help='The number of times to repeat each step')

parser.add_argument('--opts',
                  default='',
```

```
                    metavar='httperf-opts',
                    help='Options to pass to httperf')

args = parser.parse_args()

if not os.path.exists(args.bid):
    os.makedirs(args.bid)

for rate in range(args.start_rate,
                 args.max_rate + args.step_rate,
                 args.step_rate):

    # Open the log file for the rate in append mode.
    f = open('{}/{}.log'.format(args.bid, rate), 'a')

    for run in range(0, args.repeat):
        subprocesses = []

        for client in range(0, args.clients):
            rclient = clienthost.format(client + 1)
            rcmd = cmd.format(bid=args.bid,
                              client=client,
                              clients=args.clients,
                              conn=rate * 10,
                              opts=args.opts,
                              port=args.port,
                              rate=rate,
                              server=args.addr)

            execcmd = "ssh {0} '{1}'".format(rclient, rcmd)
            print (execcmd)
            sp = subprocess.Popen(execcmd, shell=True, stdout=subprocess.PIPE)
            subprocesses.append(sp)
```

```
        for sp in subprocesses:
            sp.wait()
            out, err = sp.communicate()
            f.write("\n-------\n\n")
            f.write(out)

        print("\7Press to continue")
        raw_input()

    f.close()
```

Listing 9.2: The benchmarkrunner Python script.