Toward Least-Privilege Isolation for Software

Andrea Bittau

Submitted to the Department of Computer Science in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
at the
University College London

November 2009

I, Andrea Bittau, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.

Petr Marchenko identified vulnerabilities in weak Apache/OpenSSL partitioning schemes relating to oracles and man-in-the-middle attacks. He contributed designs to combat these attacks, and realized an Apache/OpenSSL sthread implementation for his new designs.

Abstract

Hackers leverage software vulnerabilities to disclose, tamper with, or destroy sensitive data. To protect sensitive data, programmers can adhere to the *principle of least-privilege*, which entails giving software the minimal privilege it needs to operate, which ensures that sensitive data is only available to software components on a strictly need-to-know basis. Unfortunately, applying this principle in practice is difficult, as current operating systems tend to provide coarse-grained mechanisms for limiting privilege. Thus, most applications today run with greater-than-necessary privileges. We propose *sthreads,* a set of operating system primitives that allows fine-grained isolation of software to approximate the least-privilege ideal. sthreads enforce a *default-deny* model, where software components have no privileges by default, so all privileges must be explicitly granted by the programmer.

Experience introducing sthreads into previously monolithic applications—thus, partitioning them—reveals that enumerating privileges for sthreads is difficult in practice. To ease the introduction of sthreads into existing code, we include *Crowbar,* a tool that can be used to learn the privileges required by a compartment. We show that only a few changes are necessary to existing code in order to partition applications with sthreads, and that Crowbar can guide the programmer through these changes. We show that applying sthreads to applications successfully narrows the attack surface by reducing the amount of code that can access sensitive data. Finally, we show that applications using sthreads pay only a small performance overhead. We applied sthreads to a range of applications. Most notably, an SSL web server, where we show that sthreads are powerful enough to protect sensitive data even against a strong adversary that can act as a man-in-the-middle in the network, and also exploit most code in the web server; a threat model not addressed to date.

To Antonio Tundo...

Acknowledgments

I suppose that it all started thanks to **Claudio Bittau**, when I first began to copy programs from his BASIC books to the Commodore he gave me. **Dora Nikolova** planted the first seeds of my career by placing me in one of the best schools, that virtually made English my mother tongue, and that connected me to a network of people that would later shape the turning points in my life. **sorbo** bootstrapped my computer science education by giving me access to a wide range of systems to learn from. My early days at university would have been difficult without **Polina Bittau**, who made me settle with ease as I moved to my new city. I was blessed by the **Leone** and **Previti** families, who acted as my guardian angels throughout the period that followed. I then met **the man**, a professor, who is my inspiration to this day, and the reason why I chose this path in life. **Nelly Nikolova** strengthened my confidence in trying to become a professor. **Cecilia Mascolo** then discovered me, plucking me from the crowd of students, introducing me to whom would later become my PhD advisor. She also revealed to me the more "human" aspects of professors, and became a great friend. I started my PhD with **Mark Handley**, a man who taught me how to think, work, and aim for quality. His guidance constantly improved my work and myself, and I owe him for most of my maturity. **Brad Karp** supervised me too, and I admire him as the person who writes, talks, and explains most beautifully; I hope to have inherited some of those qualities. Apart from shaping much of my work and persona, he also laid out the stepping stones for my future career. **Petr Marchenko** later joined the team, and as an early adopter of my work, provided much feedback and invaluable insight. Unexpectedly, my student life was relatively luxurious thanks to **Anthony Finkelstein**, **Wolfgang Emmerich** and **Jim Fitzgerald**, and I am sure that **Hans Wilsdorf** appreciated this. Finally, **Alfred Di Rocco** always played an important role in my life, teaching me the beauty of music, which is perhaps what I really live for.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The news is full of reports of hackers gaining access to sensitive data, causing much damage and loss. Many of these cases involve attackers exploiting software vulnerabilities, as in the recent case of "Analyzer", a hacker that gained access to a myriad of credit card details by exploiting SQL servers, causing losses of at least $10M [40]. So how can we defend against such attacks, preventing the disclosure and corruption of sensitive information? Coming to the rescue is an old idea, the *principle of least privilege* [59]:

> *Least privilege: Every program and every user of the system should operate using the least set of privileges necessary to complete the job. Primarily, this principle limits the damage that can result from an accident or error. It also reduces the number of potential interactions among privileged programs to the minimum for correct operation, so that unintentional, unwanted, or improper uses of privilege are less likely to occur. Thus, if a question arises related to misuse of a privilege, the number of programs that must be audited is minimized. Put another way, if a mechanism can provide "firewalls," the principle of least privilege provides a rationale for where to install the firewalls. The military security rule of "need-to-know" is an example of this principle.*

Applying least privilege to an SQL server, for example, would prevent attackers from disclosing arbitrary database content (*e.g.,* credit cards of others), since least privilege would, in principle, limit the attacker's view of the database. But what is the "least privilege" for an SQL server in practice? To operate correctly, the SQL server needs to access, at least, its own database files. If we grant these privileges to the server, a successful attacker that exploits the server will gain these privileges too, and will therefore have the ability to disclose the entire database. This is not

the theoretical least privilege as intended by Saltzer [59], though it may well be the "least privilege" that an SQL server might need in order to run in practice in a standard UNIX environment.

The main difficulty in applying least privilege in practice is the granularity at which we can assign privilege. If we can only assign privileges at application granularity, only when the application starts, then we have no choice but to give the application all the privileges it will ever need. If, instead, we partition the application into smaller multiple compartments, we may be able to assign to each compartment only a subset of the total application privileges, because individual compartments do less work than the entire application, and hence may require less privileges. After partitioning, only some compartments will run with high privileges. These constitute the attack surface, as the attacker will need to exploit these in order to obtain high privileges. By compartmentalizing, the attack surface is narrowed to privileged compartments only. Unfortunately, even by using this approach, we can never achieve least privilege, as there are practical limitations as to how long we can continue splitting applications into finer-grained components in order to assign even fewer privileges. Least-privilege hence remains a theoretical ideal. The problem of protecting sensitive data therefore becomes: how can we *approximate* least privilege in order to make successful attacks *less probable*?

A programmer frequently has a good idea about which data manipulated by his code is sensitive, and a similarly good idea of which code is most risky (typically because it handles user input). So why do so few programmers of networked software divide their code into minimally privileged compartments? As others have noted [33, 18], one reason is that the isolation primitives provided by today's operating systems grant privileges by default, and so are cumbersome to use to limit privilege.

Our thesis is that a few simple operating system primitives specifically designed for partitioning could allow programmers to apply the principle of least privilege in their applications. We want these to be applicable to existing legacy code, without requiring applications to be totally rewritten from scratch. To facilitate their use in this context, tools could be used to determine how and where to create compartments within existing applications.

We now examine why existing operating system abstractions are not adequate for least-privilege partitioning. Consider the use of processes as compartments, and the behavior of the `fork` system call: by default a child process inherits a clone of its parent's memory, including any sensitive information therein. To prevent such implicit granting of privilege to a child process, the parent can scrub all sensitive data from memory explicitly before calling `fork`. But doing so is brittle; if the programmer neglects to scrub even a single piece of sensitive data in the parent, the

child gains undesired read privileges. Moreover, the programmer may not even know of all sensitive data in a process's memory; library calls may leave behind sensitive intermediate results.

An obvious alternative is a *default-deny* model, in which compartments share no data unless the programmer explicitly directs so. This model avoids unintended privilege sharing, but the difficulties lie in how precisely the programmer can request data sharing, and how he can identify which data must be shared. To see why, consider as an example the user session-handling code in the Apache web server version 1.3.19. This code makes use of over 600 distinct memory objects, scattered throughout the heap and globals. Just identifying these is a burden. Moreover, the usual UNIX primitives of `fork` to create a compartment, `exec` to scrub all memory, and inter-process communication to share only the intended 600 memory objects are unwieldy at best in such a situation. For example, Provos *et al.* had to jump through many hurdles to privilege separate OpenSSH using traditional UNIX APIs, as the authors had to implement their own fine-grained shared memory mechanisms and IPC infrastructure [57].

1.1 The problem

The main question we therefore want to answer is: what primitives would allow programmers to apply the principle of least privilege to their applications? These primitives should not be geared to a specific application or class of applications, but rather should be usable across a wide range of applications. As we start answering this question we quickly discover that the problem is broader. Equally important, we also need to answer the following question: what primitives and tools are necessary for applying the principle of least privilege to *existing* applications? Trying to modify existing code to increase the isolation of its sensitive data is a remarkably difficult task since the programmer needs to understand all data dependencies within a monolithic application in order to respect them when separating the application into multiple compartments. Finally, a question of ease of adoption remains: can we provide these security primitives without any changes to the operating system? In other words, can we provide an efficient implementation of our primitives in userspace, hence making our system largely platform-independent? We thus explore the role of the operating system in our security model—are OS changes necessary to provide fine-grained isolation between compartments, or are OS changes needed for increased performance, or can we avoid all OS changes?

1.2 Threat model

When designing our system, we assume the following threat model:

- Application developers are not malicious. We are not tackling the problem of safely running untrusted code.
- The operating system cannot be exploited. We limit our attention to preventing harm from vulnerabilities in application code.
- Applications may include "special" compartments that we may assume cannot be exploited. These are often the means for protecting sensitive data, and we must rely on some exploit prevention scheme for securing them. The strength of protection we provide will depend in part on minimizing the size of the code that must run in these compartments.
- Attackers can fully exploit all other compartments and we assume that arbitrary code can be run in them.
- We ultimately rely on the programmer to partition his code correctly. Although code can include exploitable bugs (in most compartments), the application must be compartmentalized correctly and compartments must have correct permissions, *i.e.,* permissions minimize compartment privilege.

1.3 Evaluation metrics

We shall evaluate our work using the following metrics:

Attack surface. Have we successfully narrowed the attack surface in applications? Does less application code have privilege to access sensitive data?

Correct application partitioning. Have we partitioned applications correctly to adhere to least-privilege, or are our designs vulnerable? We determine the information an attacker learns from exploiting untrusted compartments and verify that it does not contain any data we identified as sensitive.

Trial attacks. Can we exploit our sthread implementation or applications? Can we find any security holes? We check whether sthread protection can be bypassed (via system calls, memory accesses), and list those actions that an attacker is able to execute, if an untrusted compartment is exploited.

Applicability. Can our mechanisms be applied to a broad range of applications, such as different types of servers, and clients?

Software changes required. Can we apply our mechanisms to existing applications with only few changes to the code? Can we implement our mechanisms in existing operating systems with only a few changes?

Run-time overhead. Does our system incur a low performance cost, outweighed by the isolation gains?

Memory overhead. Does our system scale within practical limits, without running out of memory?

1.4 Contributions

We present *sthreads,* a set of primitives that allow the creation of compartments with default-deny semantics, and thus avoids the risks associated with granting privileges implicitly upon process creation. To abbreviate the explicit granting of privileges to compartments, sthreads offer a simple and flexible memory tagging scheme, so that the programmer may allocate distinct but related memory objects with the same tag and grant a compartment memory privileges at a memory-tag granularity. We contribute a full sthread implementation for the Linux kernel, showing that only a few OS changes are necessary to provide support for isolating applications. Furthermore, we also contribute a userspace library implementation of sthreads, making them easy to deploy, though at some performance cost.

When securing a complex, legacy, monolithic application, a compartment may require privileges for many memory objects, so we importantly contribute *Crowbar,* a pair of tools that analyzes the run-time memory access behavior of an application, and summarizes for the programmer which code requires which memory access privileges.

Our work led to the following insight about the interplay between default-deny semantics and tools for partitioning code: neither the primitives nor the tools alone are sufficient. Default-deny compartments demand tools that make it feasible for the programmer to identify the memory objects used by a piece of code, so that he can explicitly enumerate the correct memory privileges for that code's compartment. Conversely, run-time analysis reveals memory privileges that a programmer should consider granting, but cannot enumerate those that should be denied; it thus fits best with default-deny compartments. The synergy between the primitives and tools is what yields a system that provides fine-grained isolation, yet is readily usable by programmers.

To demonstrate that sthreads allow fine-grained separation of privileges in existing complex monolithic applications, we apply the system to a range of client and

server applications. Most notably, we apply sthreads to the SSL-enabled Apache web server, the OpenSSH remote login server, and the Firefox web browser for Linux. Using these applications, we demonstrate that sthreads can protect against several relatively simple attacks, including disclosure of an SSL web server's or OpenSSH login server's private key by an exploit, or the disclosure of sensitive information in Firefox, such as cookies, by an exploit born in the PNG image decompression library. All of this protection can be achieved while still offering acceptable application performance. We further show how the fine-grained privileges sthreads support can protect against a more subtle attack that combines man-in-the middle interposition and an exploit of the SSL web server. Regarding this latter attack, we believe that we are the first to explore the security of a cryptographic protocol in the presence of an eavesdropper that can also partially exploit the server and therefore obtain extra protocol state not visible from the network alone. We in fact prevent the attacker from recovering clear text even in such cases. The partitioning techniques for doing so themselves constitute a contribution.

We show that only a small amount of code needs to be changed to isolate existing applications when using sthreads. Most of these changes are guided by Crowbar, a tool that greatly contributes to the much unexplored subject of how to partitioning existing code. Finally, sthreads demonstrate that process-like isolation is a viable solution for isolating software, despite this approach being dismissed in the past for fear of low performance and high memory overhead [32, 18].

Chapter 2

Background

We now detail techniques used by attackers to compromise software and justify our choice of arms for defending against attacks. We describe related work and identify differences in the approach we take in this thesis, and point out where our work is complementary.

2.1 Information disclosure and corruption

We list some ways in which attackers may gain access to sensitive information:

1. Bad system configuration. An example was a flaw in VMware's configuration script that installed an SSL key with weak permissions, allowing local users to read the SSL private key used for encrypting management and console traffic [10]. This ultimately allowed attackers to access the VMware server, by decrypting eavesdropped management or console credentials.

2. Race condition. An example was a flaw in fetchmail's configuration utility that wrote the configuration file before restricting the file's access [78]. An attacker could have therefore tried to read the configuration file just after it were written, but before its permissions were changed. This configuration file contained cleartext usernames and passwords, giving the attacker full access to the victim's e-mail account.

3. Program logic error. An example was a flaw in Windows' remote share authentication, that allowed attackers to authenticate by supplying the correct first character of the password [50]. One can imagine that the code did a comparison based on the length of the supplied password, rather than on the length of the stored password. This allowed attackers to access remote shares by brute-forcing only the first character of the password, and to incrementally

discover the password, as they now had an oracle for the password's prefix (the attack works for subsequent password characters, too).

4. Command injection. An example was a flaw in sendmail, that allowed an attacker to pass a command into `popen` [45]. Sometimes programs execute external programs passing user input as parameters. By inappropriately escaping and sanity checking this input, attackers may insert commands to execute. This allowed attackers to execute arbitrary commands on the remote server.

5. Application memory corruption exploit. An example was a stack overflow in fingerd, that allowed an attacker to execute arbitrary code [46]. By not providing enough buffer space for user input, it is possible for such input to overwrite memory in a program. This allows attackers to inject code into memory, and overwrite vital program structures in memory, ultimately causing the program to jump to injected code. Attackers can therefore execute arbitrary code in the context of the exploited application.

6. Virtual machine bypass (or exploit). An example was Microsoft's Java bytecode verifier, that failed to check for some occurrences of malicious code [44]. This allows attackers to execute unsafe code, despite the client believing that all code is sandboxed.

7. Kernel exploit. An example was Linux's `brk` vulnerability, that allowed userspace applications to access kernel memory [67]. This allowed local attackers to, for example, change the userid of their currently running process to root.

8. Covert channels. An example was OpenSSH's root login timing attack, that allowed attackers to determine whether their current guess for the root password was correct, by inspecting response time, when root logins were disabled [7].

Solutions often vary in the range of attacks they defend against. For example, writing applications using safe languages like Java prevents any memory related exploits. Java, however, would not help against filesystem race condition bugs. These, instead, could be prevented by a mandatory access control system such as SELinux [41], by installing a policy that denies applications from writing unclassified files. To protect against even more classes of exploits, like kernel ones, one may need to adopt a different solution, like Asbestos [18] and HiStar [82], which minimize the size of the trusted kernel, reducing the targets available to an attack. Unfortunately, some classes of vulnerabilities, like program logic bugs, may have no cure apart from manual code audit, since the vulnerability has been "legitimately" implemented as a "feature" of the software, making it difficult for protection schemes to detect attacks

and stopping them. It is typical for solutions to tackle some classes of attacks, but not all. It is possible, however, to use a set of solutions in combination, like Java and SELinux, to protect against broader ranges of attacks.

Apart from range of attacks covered, solutions also vary in complexity of adoption. For example, supporting mandatory access control requires extending existing operating systems. Using safe languages requires rewriting all applications in a new language. To protect the kernel too, Asbestos and HiStar required rewriting the entire OS. This varying complexity also seems to have a relationship with the attack coverage of a solution. For example, Asbestos and HiStar cover a wide range of attacks, including race conditions, command injection, application exploits and kernel exploits. Unfortunately, the solution's adoption complexity is high as it requires an entire new OS. Systems like SELinux, instead, have lower adoption complexity as they are an extension (rather than rewrite) to operating systems. SELinux though, protects against a much narrower set of attacks compared to Asbestos and HiStar, allowing, for example, application exploits and kernel exploits.

Our goal is to build a readily deployable solution that covers an important class of attacks, specifically, application exploits. In 2008 alone, according to the Open Source Vulnerability Database [52], 1,290 vulnerabilities were reported under this class—more than three a day. The impact of these vulnerabilities is high, as some are remotely exploitable, and some target widespread products, such as Microsoft Office or the Apache web server. Compare this with kernel exploits, where only 108 have been reported, though such vulnerabilities typically have much higher impact (*e.g.,* guaranteed root access). Protecting against kernel exploits likely requires a high complexity solution to minimize the trusted kernel size (like HiStar), so we refrain from tackling this problem as we seek a simpler to deploy solution. Instead, we focus on application exploits which still exist in significant numbers. To protect against even more attacks, like race conditions and command injection, we use existing solutions like SELinux (already deployed) in combination to ours. The cost for the simplicity of our solution is that it will fail to protect against bad system configuration, program logic errors, kernel exploits and covert channels. Referring to the list of attacks presented at the beginning of this section, our complete system will thwart the following categories of attacks: 2, 4, 5, 6. The last two are of particular focus to our work as they cannot be solved alone by the existing deployed systems we leverage (*e.g.,* SELinux) and thus represent our main contribution.

2.2 Partitioning and least privilege

So how do we protect against application memory-related exploits? One general solution is to use a safe language. The adoption complexity of this solution though is high, as it requires rewriting all applications in a new language. We desire a simpler to adopt solution which can be used in the short run.

There are a number of less general but simple-to-use point solutions to prevent application exploits. For example, writable or executable (W⊕X) memory [51] prevents injecting runnable code into an application, which is a technique typically used to run arbitrary code. While this is a simple solution to adopt, it is vulnerable to return-to-libc attacks [64]. Used in conjunction with address space layout randomization [70] though, it can be used to combat return-to-libc attacks. This is not a permanent solution though, as address space layout randomization implementations have been shown to be broken [61], and there have been bugs in the past which leak information revealing memory addresses [65]. We therefore seek a general solution for application memory-related exploits, more like safe languages, though with low adoption complexity, more like W⊕X.

Rather than preventing exploits from occurring (like W⊕X), an alternative strategy is to contain exploits, by giving code the least privilege it needs to execute. Thus, a successful exploit will only gain least privilege, which may not be sufficient to disclose sensitive information. A realization of this is privilege separation, applied to OpenSSH [57]. In privilege separated OpenSSH, a network client is handled by two processes: a monitor and child. The monitor is privileged and consists only of a small fraction of the entire code. The bulk of the implementation lies in the unprivileged child. If vulnerabilities occur evenly in code, the child is more likely to contain them as it is larger. Thus, it is more likely that attackers will exploit the unprivileged child, and not gain access to any sensitive information, such as passwords, which are shielded by the monitor. Having a small monitor also makes it more feasible to manually audit its code, to remove vulnerabilities. Privilege separated OpenSSH has been a success as no public memory corruption exploits work on it, though there are exploits which work on non-privilege-separated OpenSSH [80, 27].

Partitioning and least privilege help in two main ways:

1. They minimize the amount of privileged code. Since a smaller fraction of code is privileged, it is more feasible to manually audit this code to eliminate vulnerabilities. If vulnerabilities are distributed evenly in code, then there will be less of them in the smaller privileged code.

2. “Riskier” code can be run in unprivileged compartments. Vulnerabilities are

not distributed evenly in code, but rather, past exploits show us that problems typically occur when handling user input. For example, in 2008 alone, 840 vulnerabilities have "parsing" or "handling" in the title as reported by OSVDB. Even OpenSSH's challenge/handshake vulnerability [27] involves reading larger-then-expected input from the network. Thus, handling foreign (*e.g.,* network) input seems a "risky" operation, so running such code in unprivileged compartments will have real benefits. Similarly, having privileged compartments take more restricted inputs can minimize their risk of being exploited as the attacker will have less leverage, and fewer (and restricted) inputs are simpler to sanity check prior to extensive, error-prone, manipulation.

Unlike privilege separation in OpenSSH, we intend to explore generic mechanisms for partitioning applications, without being tied to the monitor/child model, allowing for even more fine-grained partitioning.

2.3 Complexity in partitioning

Partitioning and least privilege provide a general solution for tackling application exploits. The drawback of such partitioning is the assumption that privileged compartments remain unexploited. The benefit of partitioning is that it is less complex than a language based solution. Partitioning requires restructuring an application—not rewriting it in a different language. Note that an interesting combination of partitioning and language based solutions would be to implement trusted compartments in a safe language. Since such compartments are typically small, the effort of rewriting them in a different language may not be great. This would remove the assumption that privileged compartments must be unexploitable, since safe languages provide this property.

But how simple is it to restructure applications and to partition them? Is it really a low complexity solution? The authors of privilege-separated OpenSSH had to surmount many problems in order to implement their solution: for example, they had to implement their own fine-grained memory sharing and IPC mechanisms [57]. The authors of OKWS too lament that fine-grained isolation with UNIX primitives is difficult [32]. We shall note that primitives specifically geared for partitioning do simplify the problem, but it still remains remarkably difficult to partition *legacy* code. In legacy code, all code has the privileges of the whole application. When partitioned, this property no longer holds and the application breaks due to privilege violations. To retain the property of ease of adoption, we need a mechanism for partitioning legacy code.

In the past, this problem of how to partition legacy code has been tackled, for example, by Privtrans [14], which privilege separates legacy applications automatically, according to the monitor/child model. We wish to generalize privilege separation to allow more fine-grained partitioning, where an arbitrary number of compartments can be created. To retain ease of adoption, like Privtrans, we need to explore a mechanism for applying our primitives to legacy code.

2.4 Related work

We divide related work into two main categories:

1. Mechanisms for least privilege partitioning of applications. We compare these works to sthreads.

2. Tools for privilege separating legacy applications. We compare these to Crowbar.

2.4.1 Least privilege partitioning mechanisms

Many ideas present in sthreads come from capability systems [77, 79, 63, 9, 19]. Both in capability systems and sthreads, each protection domain (*e.g.,* an sthread) can grant a subset of its privileges (*e.g.,* memory privileges, file descriptors, *etc.*) to another protection domain. Capability systems demonstrate how an entire system and its security can be based around the single concept of a capability. Such pure capability systems require a clean-slate OS design in order to build all system abstractions from capabilities, and to track and enforce the spread of capabilities throughout the system. sthreads do not require a clean-slate OS, but rather minimally extend existing OS abstractions, adding simple isolation features that give a good tradeoff between system complexity and security benefit. Unlike capability systems, sthreads do not offer any kernel protection as the kernel is not rewritten in terms of them. Capability systems define a programming model where all system resources are accessed via capabilities, and where programs must expose the various services they implement as capabilities, granting them to other programs as needed. sthreads instead are designed to simplify the isolation of legacy code, thus support all standard UNIX calls, and additionally expose APIs that more closely resemble abstractions already used today, like memory allocation and thread creation. Capability systems could certainly implement sthreads to simplify isolating existing code, while retaining all the security benefits of running on a full capability system.

Callgates (privileged entry points) in sthreads were inspired by gates in Multics [60] and x86 [25]. One difference is that callgates in sthreads are not hierarchi-

cal, and also allow a richer set of privileges to be associated with them, such as file descriptor permissions and a system call policy.

sthreads resemble variable weight processes [6], in that they lie somewhere in between processes and threads in terms of complexity and shared information. Unlike variable weight processes though, sthreads allow finer grained control over shared memory, file descriptors and allowed system calls, critical to security.

Opal [15], too, supports cross-thread sharing of memory regions. But Opal does so in a single-address-space OS, with a modified compiler that statically places text and data in the OS's global address space. Opal was designed for memory-sharing-intensive applications, such as CAD systems, rather than to allow fine-grained isolation for least privilege.

Asbestos's event processes [18], like sthreads, are a lightweight mechanism for executing with different memory privileges. Unlike event processes, sthreads allow concurrency, though at the cost of much higher memory consumption. Thus, sthreads resemble traditional threads more closely, whereas event processes mimic tasks in event driven programming, the latter being a cheaper mechanism with the limitation that only one task can run at a time. By allowing multi-threading, sthreads are more general than event processes, as the latter are restricted to a programming model where only one event process can run at a time.

sthreads are complementary to MAC [17] systems, and indeed use SELinux [41] to protect the interactions between an sthread and the rest of the system. sthreads' memory and file descriptor protection allow specifying policies for interactions *within* a single application, impossible with existing MAC systems like SELinux.

sthreads, like Nooks [68], create a lightweight protection domain. Unlike Nooks, sthreads tackle userspace applications rather than the kernel. Nooks' focus is on integrity, and requires additional complexity to support unmodified code. Nooks must track any memory changes so that it can recover state on errors. sthreads' integrity guarantees are much simpler than Nooks': either an sthread can or cannot write to memory—there is no functionality (nor need) for reverting memory changes after writes. Being a security system, sthreads also provide privacy, unnecessary in Nooks, as Nooks' goal is to provide fault tolerance.

sthreads take a different approach from decentralized information flow control (DIFC) systems. Asbestos [18], HiStar [82] and Flume [34], allow untrusted code to compute over sensitive data. The system is responsible for tracking information flow and disallowing data to be exported from the system, unless authorized by a privileged process. sthreads are not flexible enough to accommodate this model, as they require that sensitive data is manipulated only by privileged compartments. To track information flow, Asbestos and HiStar require a new OS (unlike sthreads).

By having a small kernel, these systems are also more robust against kernel exploits. Flume, instead, runs on Linux, so like sthreads, can be readily adopted.

The approach to security taken by sthreads is directly inspired from privilege separation in OpenSSH [57] and OKWS [32]. In both OpenSSH and OKWS, applications are split into multiple processes in order improve isolation and reduce privilege needed by each process. Both OpenSSH and OKWS though use mechanisms specifically tailored for those two applications. sthreads instead provide a generic framework for privilege separating applications. Privman [30], like sthreads, is a generic mechanism for privilege separating applications. Unlike sthreads though, it is tied to the monitor and slave model, taken from OpenSSH. sthreads instead allow creating an arbitrary number of compartments.

sthreads focus on containing memory-related exploits. Safe languages cure these problems as memory corruption cannot occur with them. With sthreads, we rely on some compartments being privileged, and a successful exploitation of such compartments would give away privilege to the attacker. Languages such as Java therefore provide stronger guarantees since they protect the entire application. The advantage of sthreads is that they can be applied to legacy C code, and if performance is paramount, native code runs faster than interpreted code. Safe languages can be complementary to sthreads, by implementing trusted compartments in sthreads with safe languages.

Jif [47] uses static analysis on Java to support information flow control. With Jif, programmers can guarantee that code adheres to a specific information flow control policy. For example, they can prevent a method from returning sensitive information. Jif's analysis is sound, meaning that if the compiler is implemented correctly, then Jif *guarantees* that only the explicitly allowed information flow will ever occur at run-time. Such strong guarantees cannot be provided by sthreads and hence sthreads are not sound. With sthreads, programming errors that occur in a privileged compartment that, for example, leak (or copy) sensitive data into a memory buffer shared with an unprivileged compartment may cause sensitive information disclosure. Such errors are prevented by the Jif compiler. Hence sthreads are weaker as they rely on the programmer's partitioning an application correctly and auditing privileged code for correctness. We note that both Jif and Coyotos require specific languages (respectively Java with annotations and BitC) to allow formal verification, a property which may be in opposition to one of sthreads' main goals of wanting to support legacy C code. Finally, because sthreads are not a DIFC system, they do not allow tracking information flow, which is instead done by Jif.

SFI [75] enforces memory protection in software. Though SFI's main goal is fault-isolation, the mechanism can be used for security, too. The same memory

isolation can be achieved with sthreads and tagged memory, but by using hardware page-protection instead. SFI provides memory protection in software by adding checks to object code, and limiting what memory code can write or read. Additional checks are necessary to prevent code from jumping to an arbitrary location, in an attempt to skip checks. Implementing this on x86 is a challenge due to variable length instructions, and more control flow checks would be needed to thwart attacks like return-to-libc [64].

CFI [4] provides control flow integrity, necessary for enforcing security with SFI, and does so on x86. sthreads take a different approach compared to CFI (and SFI): sthreads do not *prevent* exploits, but rather *contain* them. With sthreads, exploits are allowed to occur in unprivileged code as such code has no access to sensitive data and cannot therefore disclose it (one of our security goals). With CFI, the exploit would be prevented from occurring altogether, satisfying the same end goal. However, CFI is less general than sthreads as it only prevents exploits of a specific category—those that alter the control flow of a program (which admittedly, is a high proportion of exploits). What if the exploit flips a vital bit in the program that bypasses authentication, or there is a format string vulnerability [36] that discloses sensitive data without altering the program's control flow? These are not stopped by CFI but can be stopped by sthreads, if the vulnerability occurs in unprivileged code. Of course sthreads provide no protection against vulnerabilities in privileged code, where CFI instead could. In fact, the two systems can be used in conjunction to provide higher levels of security.

XFI [20] combines SFI [75] and CFI [4] to isolate code and prevent exploits. It is stronger than CFI as it can control the memory available to code, and thus thwarts attacks that alter (or disclose) vital program data. It has been used in practice in drivers, where predictable memory regions are accessed. It is less clear how to apply these techniques, in general, like sthreads, to split an application into an arbitrary number of compartments, each of which can access a large number of different memory regions, as dictated by an arbitrarily complex policy. XFI can be complementary to sthreads to protect callgates (which we assume are unexploitable), where the memory interface (*i.e.,* arguments) to callgates is well defined and typically minimal. XFI too may benefit from sthreads as the high expense of XFI will be paid only for a small fraction of the code (callgates).

Virtual machines are also being used for software isolation [74, 55]. However, they are typically used for coarse-grained partitioning, such as providing multiple OS instances on a single box, or running a VPN client or router in a separate VM. Though the isolation of VMs is strong, the scalability of sthreads is greater. Furthermore, because of better scalability, sthreads enable partitioning applications

at a finer-granularity to give components tighter permissions in attempt to increase overall security. Such fine-grained partitioning may be impossible with VMs for scalability reasons. sthreads may however be able to leverage VMs for callgates. Since there are typically only a few callgates per application, and their security is most important, it may be possible to run callgates in a VM, while running the (many) sthreads in a different VM, to further improve isolation of callgates.

TaintCheck [48] prevents the execution of malicious code. All user input is "tainted" (marked) and its flow tracked through the system and all such data is prohibited from being executed. To implement this, the application's code is heavily instrumented causing a large performance degradation, making the system impractical for high performance servers. To gain better performance, hardware based solutions have been proposed, like that by Ho *et al.* [24]. Even with hardware support though, performance remains good only when little tainted data is present.

Static analysis has been used to detect security violations in programs, such as finding format string vulnerabilities [62]. When applied to C, these approaches examine programs for known attack techniques. Run-time mechanisms like sthreads can be used as a complement since they will contain both known and unknown memory-related attacks, rather than trying to prevent known attacks only.

2.4.2 Tools for privilege separating applications

Privtrans [14] uses programmer annotations and static analysis to automatically split applications into two processes. Crowbar, our tool for assisting in application partitioning, is more limited in two ways. First, it is a manual process, which arguably requires more work. Second, being a run-time tool, it suffers from a coverage problem, as multiple runs with different inputs may be required to yield all necessary privileges of the program being studied. Privtrans is limited to the monitor/slave model, as used in OpenSSH, whereas Crowbar enables programmers to partition applications in arbitrary ways.

Jif/Split [81] also automatically partitions applications via static analysis, though it is targeted for Java programs. Crowbar targets legacy C code (and is run-time).

Scrash [12] uses programmer annotations and static analysis for identifying sensitive data, and allocating this memory at run-time in protected regions. Upon a crash, any sensitive data and protected regions can be removed from core files, so that core files can be sent to untrusted parties for debugging. Crowbar too, requires the programmer to list sensitive data, though it uses run-time analysis to determine the propagation of sensitive data. In addition, Crowbar also points out which code uses sensitive data to identify callgates. Scrash only needs to isolate sensitive data.

Program slicing techniques [72] can statically provide data dependencies in software. The focus of these approaches though has been for debugging, rather than providing information for splitting applications into multiple compartments. Because program slicing can be performed statically, it may complement Crowbar's output, by providing information that may not have been obtained by a particular run.

2.5 Summary

On commodity UNIX systems, programmers merely have coarse-grained mechanisms such as `setuid`, `chroot` and `fork` to partition and secure their application. We propose primitives for fine-grained isolation and provide a userspace implementation of these which can be readily used in commodity OSes. Much of the previous work required either newly written OSes, coarse-grained partitioning, or mechanisms specifically geared towards one application.

We tackle the problem of partitioning legacy code and provide tools that help programmers in doing so. There has been little work in the past on tools for splitting legacy C applications in a generic way for security purposes.

Chapter 3

Primitives for securing applications

We now show conceptually how one can use partitioning to secure an application. We then apply these concepts to the C programming language and UNIX operating system, defining our *sthreads* primitives. We discuss sthreads in detail, validate their security properties by running trial attacks, and describe limitations.

3.1 Concepts required for partitioning

Our goal for least-privilege partitioning is to narrow the attack surface to compartments that run a small fraction of the application's code, that perform simple, well defined operations, with well defined interfaces. A trusted component that performs complex network parsing is unacceptable as writing such code is error-prone, and inputs can be arbitrary as read from the network, thus more difficult to sanity check. On the other hand, a trusted component that traverses a linked list, searching for a fixed length key is acceptable. This code is likely to be small in size, and thus can be more easily audited. Its inputs are well defined, making any sanity checks on them easier. Objectively, therefore, our goal is to minimize the amount of code that

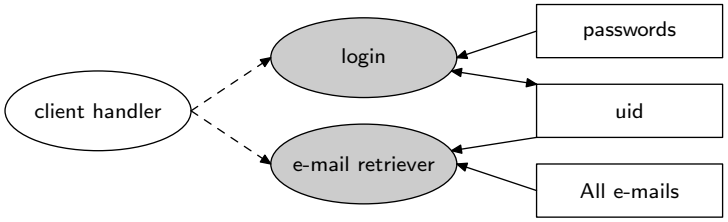


Figure 3.1: Partitioning of a POP3 server.

must run in trusted compartments. More subjectively, the operations performed by these components must be simple and their interfaces (and hence inputs) limited, and well defined.

To make least-privilege partitioning a bit more concrete, consider a toy example of how one might partition a POP3 server, as depicted in Figure 3.1. Ovals represent compartments, where shaded ones are privileged; rectangles represent memory regions; arrows represent read permissions (read/write if double arrow) and dashed arrows the ability to use or invoke another compartment. We do not show system objects such as files as our work focuses on protecting memory; we use existing solutions, namely SELinux [41], to protect system objects. Similar SELinux protection needs to be applied to files: *e.g.,* only the login compartment should have read access to the `/etc/passwd` file. One can split the server into three logical compartments: a client handler compartment that deals with user input and parses POP3 commands, a login compartment that authenticates the user, and an e-mail retriever compartment that obtains the relevant e-mails. The login compartment will need access to the password database, and the e-mail retriever compartment will need access to the actual e-mails; these two are *privileged* in that they must run with permissions that allow them to read these data items. The client handler, however, is a target for exploits because it processes untrusted network input. It runs with none of these permissions, and must authenticate users and retrieve e-mails through the restricted interface to the two privileged compartments. Although this partitioning approximates least privilege, it is not least privilege in the absolute sense. For example, the e-mail retriever has access to *all* e-mails. With least privilege, it would have access to only the particular e-mail that will be retrieved. From the client handler's perspective though, its privileges approximate least privilege as the client handler will only obtain the e-mails it is allowed to read, if the other trusted components remain unexploited.

Because of this partitioning, an exploit within the client handler cannot directly reveal any passwords or e-mails, since it has no access to them. Note that this partitioning is instantiated once for every client. Thus, from the network, the client handler will only be reading passwords for the particular connection being handled. That is, by exploiting the client handler, the attacker will only be able to read his own network traffic. We program the e-mail retriever to only read e-mails for the user id specified in *uid*, which can only be set by the login component. Thus, an attacker cannot retrieve e-mails by skipping authentication since *uid* must be set, and this is only set after successful authentication via the login component. For the above arguments to to hold, one must, however, ensure that code running in the privileged compartments cannot be exploited. So long as there is more code in

the client handler than in the privileged compartments, partitioning has successfully narrowed the attack surface. Note that the client handler will contain risky code since it needs to parse relatively complex network input. Also note that there is a well defined interface to trusted compartments, so the range of inputs can be narrowed or even sanity checked at the defined boundary. A typical monolithic implementation would instead combine the code from all three compartments into a single process. An exploit anywhere in the code could disclose anything in the process's memory, including any resident passwords and e-mails. Hence, partitioning can reduce the impact of exploits—a result from the principle of least privilege [59].

Our aim in building sthreads is to allow the programmer to create an arbitrary number of compartments, each of which is granted no privileges by default, but can be granted fine-grained privileges by the programmer. To the extent possible, we would like the primitives we introduce to resemble familiar ones in today's operating systems, so that programmers find them intuitive and minimally disruptive to introduce into legacy application code.

3.2 Applying concepts to C and UNIX

Exploits that target network servers are a major threat as they allow remote compromise of hosts, so we chose to target sthreads for UNIX and C, which are widely used to implement such servers. We therefore wish to map the abstract concepts of partitioning and protection to concepts present in C and UNIX. The result is the following three abstractions:

Sthreads. These are unprivileged compartments. When used correctly, if these are exploited, the attacker should gain no extra information beyond that which has been explicitly given to the sthread. For example, if the programmer creates an sthread that can only read a network buffer of the connection being handled, then the attacker will only be able to read his own network data—no benefit is given to the attacker as he already knew that data without having to exploit the sthread. One can think of sthreads as "default-deny processes"—*i.e.,* a `fork`-like call, where, by default, no memory, file descriptors, or any other resources are inherited from the parent.

Tagged memory. This is a mechanism for sharing (and naming) memory. Because, by default, no memory is shared between sthreads, we need a mechanism to share memory between co-operating sthreads. For example, a parent sthread may want to pass an argument to a child sthread. To do so, it must explicitly share memory for the argument and allow the child to read that

memory, hence the need for our tagged memory mechanism.

Callgates. These are privileged compartments. Callgates are a special use case of sthreads, though we name them differently to highlight that callgates (unlike sthreads) are privileged. Our APIs differentiate callgates further by providing a simple mechanism to invoke them synchronously, and different mechanisms to create callgates, compared to sthreads. Callgates are typically used to perform privileged operations in applications, such as authentication. If a callgate is exploited, the attacker gains access to its elevated privileges. Hence, we must ensure that callgates are robust and carefully audited. Fortunately, often only a very small part of an application's code need execute in callgates. One can think of a callgates as an sthread that provides a service that other sthreads can invoke. Any arguments and return values are passed via tagged memory.

We now discuss these three abstractions in detail, in the sections that follow.

3.3 Unprivileged compartments

Our goal is to split an application into minimal-privilege compartments. We therefore need a mechanism for specifying a compartment, namely, an sthread. An sthread is a "default-deny process". The sthread system call is like `fork`, although by default, the full memory map and the file descriptors of the parent are not copied to the child. Hence a newly created sthread, by default, cannot access any memory from its parent nor any file descriptors. Thus, by creating an sthread, one can be sure that by default the sthread cannot read any data from the parent, nor tamper with the parent in any way—the two are completely isolated from each other. The same applies to siblings and children. In reality, by default, an sthread will have access to some memory critical for execution. For example, it will include a private stack and heap, and read-execute access to the program's code. No "data" memory is inherited though, and this is what contains sensitive information.

To do useful work, an sthread will probably need access to some memory and file descriptors from its parent. For example, the parent may want to pass arguments to the sthread. One can pass file descriptors and memory pointers directly, through function parameters, just as one would do with threads. With traditional processes, however, one must create shared memory segments, or may need to pass file descriptors across UNIX sockets. From a usability perspective, therefore, sthreads are like traditional threads, although from a security perspective, sthreads are more like processes. Sthreads cure three deficiencies of `fork` in the context of security. First, they remove the risk of unintentionally leaking information (*e.g.,* sensitive memory)

from the parent to the child. Second, they allow easy sharing of data when necessary. Third, as discussed in Chapter 6, they are optimized to support a high creation rate, necessary for high throughput servers where a new sthread is needed for each new client.

When passing parameters to a child sthread from a parent sthread, there is an additional step involved, apart from passing (say) a pointer. The programmer must specifically allow access to that memory within the parent in the child sthread's *security policy*. This way, the programmer can control whether the memory can be accessed read-only or read/write by the child. A security policy is attached to an sthread upon creation and by default allows no memory or file descriptor access. The policy can specify the following:

- The memory that an sthread can access (read, read/write).[1]
- The file descriptors that the sthread can access (read, write, read/write).
- The callgates an sthread can invoke.
- An SELinux policy for the sthread.

The first two items allow an sthread to access particular memory and file descriptors. In policies, file descriptors are specified by their number. Memory is referenced by its *tag* rather than its base pointer. This allows multiple memory addresses to be referred to by a single tag, making policies shorter and simpler to specify. Furthermore, it allows grouping memory objects with the same tag into the same pages to allow for hardware page protection. This will be discussed in detail in the next section. An sthread can also be allowed to invoke a callgate, which typically performs a privileged operation (such as authentication) on the sthread's behalf. Finally, each sthread can have an SELinux policy attached to it that limits the system calls the sthread can invoke, controlling the sthread's access to system resources such as processes and files.

When creating a new sthread, the parent sthread can only create a child with equal or less privilege than itself. This prevents sthreads from elevating their privileges, and assures that the sthread mechanism can only be used to drop privileges. Specifically, an sthread can only create sthreads with a subset of its memory and file permissions, a subset of its allowed callgates, and an SELinux policy transition that is allowed by the global SELinux policy. The sthread policy cannot be changed over time: it is fixed at sthread creation time.

[1] We do not support write-only memory as it is not supported by page-table protection on x86 hardware. We discuss implementation details in Section 3.9.

3.4 Memory protection

We now detail how our memory protection mechanism works, and how we refer to memory in sthread policies. One way of referring to memory in sthread policies is by pointer addresses. This can easily become tedious when sharing large data structures that are created via multiple allocations. Instead, one would like a single name for the whole data structure, or even a single name for multiple data structures. To accommodate this need we introduce *tagged memory regions*. Every time a programmer allocates memory, he can specify its *tag*. He can then refer to this tag in the sthread's security policy. We provide a library function `smalloc` that takes two parameters, a size and a tag, which is simply a wrapper to `malloc`, though it allocates from a specific arena according to the tag. A programmer can still use `malloc` to allocate (private) memory although the returned objects cannot be named in sthread policies, and therefore cannot be shared among sthreads. Indeed the majority of memory typically is private to an sthread, so programmers will mostly use the familiar allocator `malloc`.

We tag memory this way for another practical reason. We use hardware page protection to enforce memory permissions for sthreads. Hence, our implementation requires that every virtual memory page contain part of the arena for at most one tagged memory region. Because we specify a tag for every allocation with `smalloc`, allocations with the same tag are placed in the same arena, whose pages all contain only data allocated with the same tag. Otherwise, we risk that memory with different tags ends up in the same page, and we would not be able to use page protection to enforce access control independently for different tagged memory regions. Once assigned, the tag of memory objects cannot be changed, though we did not find this to be a limitation as tags act purely as a naming mechanism.

3.5 Privileged compartments

Creating a new sthread only allows reduction of privilege. Callgates allow elevation of privilege. The existence of callgates thus permits creating sthreads with minimal privilege; additional privilege for isolated code fragments can be delegated to callgates. By doing so, we limit the attack surface to callgates, which we argue can be designed to be much smaller than sthreads in terms of lines of code.

When using sthreads on a network server, a common design would be to have a privileged *master* sthread spawn worker sthreads, each of which deals with one client. This way, the master is isolated from the workers, and the workers are isolated from each other—a hacker connected to one worker cannot directly attack the

master or other clients. Suppose that the network service requires authentication. If so, the worker sthread needs sufficient privilege to perform authentication; *e.g.,* it must be able to read the password database. One option would be to give the sthread sufficient privilege. This defeats the aim of least-privilege, though, since if the sthread is exploited—perhaps likely, since it may perform complex protocol parsing—then the attacker would gain access to the password database. A better option would be to give *minimal* privilege to the sthread and create a *callgate* that performs authentication on the sthread's behalf. The sthread can then be given the privilege to invoke the callgate to perform authentication. Thanks to this callgate, the programmer only need audit and secure the (likely shorter) authentication code, and not all of the complex and long protocol handling code present in the sthread. Callgates therefore provide a mechanism for factoring out privileged operations from applications, and permit sthreads to run with significantly reduced privileges, while allowing complete functionality through callgate invocation.

From a usability perspective, a callgate is just like an sthread. When created, it is assigned an sthread policy. At its creation time, an sthread can then be given privilege to invoke the callgate. When invoked, the callgate will run with the permissions that it was created with. When creating a callgate, the same rules of sthread creation apply: an sthread can only create a callgate with lesser or equal privileges to those it holds. An sthread thus cannot create an arbitrary callgate and escalate its own privileges. To invoke a callgate, though, an sthread's parent simply needs to allow the child sthread to invoke that callgate, regardless of the callgate's privileges. In sum, then, callgates' privileges are fixed upon creation and granted upon invocation. Because a callgate executes as a separate sthread, it is isolated from its caller. A malicious sthread cannot invoke a callgate and, for example, change the contents of the callgate's private memory to inject an exploit.

Because callgates are privileged, the programmer must ensure they remain unexploited. A programmer may reduce the likelihood of the exploitation of a callgate by auditing its code. Beyond code audits, a more robust sthread system could harden callgate code against exploits using such techniques as CFI [4], and the execution overhead of CFI would then only be paid for the small fraction of code that runs in callgates. Note that because callgates define an explicit boundary between trusted and untrusted code, and our sthread system enforces that callgates are entered at a known code location, it is possible to sanity-check inputs to decrease the probability of exploits. Callgates' known entry points are essential to this property; checks inserted by the programmer could be skipped by exploits without known entry points. A callgate's code can begin with argument checking in much the same way an operating system checks arguments when system calls are invoked. Even when the code

that executes within a callgate is large, the risk of being exploited can be lowered by properly checking arguments and exporting a narrow interface.

Since a callgate can only be exploited from its input—*i.e.,* its arguments, programmers must therefore assume hostile arguments and sanity-check them. We provide a mechanism, namely the *trusted argument*, which is an argument passed to the callgate from its creator. The creator of a callgate is trusted because it has complete control over the callgate anyway. The programmer need not check the trusted argument's sanity. This argument could be used to pass data whose correctness is vital for the callgate's correct execution. For example, an authentication callgate could use the trusted argument to obtain a pointer to the password database. If this pointer were obtained from the calling sthread, the sthread could spoof a password database rather than passing a pointer to the real one, and because the sthread knows the passwords in its own fake database, it can successfully authenticate via the callgate without knowing real credentials. The trusted argument therefore allows a callgate to obtain any information that must be valid for the security of the system, and to minimize the arguments that must be passed from a possibly hostile sthread.

3.6 System call protection

The mechanisms described so far (sthreads, tagged memory, callgates) are sufficient for protecting the internals of an application. That is, one can use them to (say) prevent one network client in a server from accessing the data of another client. This protection is insufficient, though, because an attacker could exploit an sthread to obtain or corrupt data beyond that stored in memory. For example, the attacker could try to read files from disk, or open raw sockets to obtain sensitive data belonging to other users. We therefore must constrain the system call privileges of sthreads.

To do so, we can use an existing mechanism, such as systrace [56] or SELinux [41], or perhaps devise our own, such as a simple bitmap of which system calls an sthread may invoke. Since our implementation is for Linux, we use SELinux. Each sthread can run with a different SELinux ID (SID), and we can therefore control which system resources each sthread can access. SELinux also has a default-deny philosophy, so it fits nicely in our framework. By default, an sthread's SELinux policy allows it to do nothing. Permissions, therefore, once again, must be explicit.

We believe that SELinux itself will benefit from sthreads as they become used, and the interplay between the two makes the whole system much more powerful. One of the problems faced by SELinux is that a process operates under only one

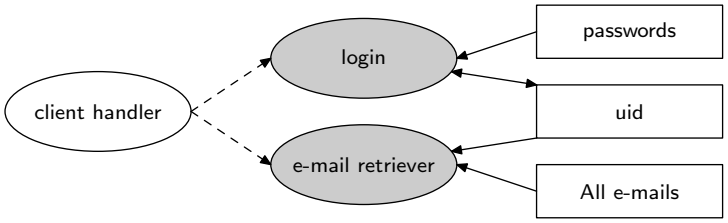


Figure 3.2: Partitioning of a POP3 server.

SELinux policy, and many applications are written in a monolithic, single-process, fashion. For example, what policy would be appropriate for a single-process file server? Since any user must be able to access the file server, the file server's policy must allow any file to be read. Such a policy offers no reduction in privilege, since a successful attacker would then have access to any file. This is a real problem for SELinux and Samba [38]. By partitioning Samba using sthreads, we can now apply SELinux policies at an sthread granularity. Likely, a partitioned Samba would serve each client in a different sthread, allowing us to apply a different SELinux policy to each client, depending on which user the client authenticates as. A successful attacker would therefore only be able to access the files owned by the account he controls, and not the whole filesystem.

3.7 Example: securing an incoming mail server

We illustrate the use of sthreads with the toy example introduced at the beginning of this chapter: an incoming mail server (POP3). Note that this is an illustrative example only; it is simpler in its partitioning than most real-world applications, which we examine in Chapter 5. Our security requirements for the mail server are:

- Each user can only access his own e-mails. After successful login, a user must be prevented from reading or modifying e-mails of others.
- Each user must authenticate before reading e-mails. A user must be unable to read e-mails by skipping authentication.

To accommodate these goals, we use sthreads as shown in Figure 3.2 (the same one introduced earlier in this chapter). White ovals represent sthreads and gray ones callgates, and rectangles are tagged memory regions; arrows indicate permissions as already described. We encapsulate all the client-handling functionality into an sthread, and create one sthread per connection. This approach ensures that connections are isolated from each other, as distinct sthreads cannot by default access one another's memory, and therefore, one user's sthread cannot (for example)

read an e-mail from the memory of a foreign `client handler` sthread. Such action would, however, be possible in an implementation in which all clients were handled by standard threads within the same process. Next, we do not grant the client handler sthread privileges to access the password database or e-mails directly. Instead, we grant these privileges to two trusted components (callgates): `login` and `e-mail retriever`. The e-mail retriever will only retrieve e-mails belonging to a specific user id (UID). The UID is a tagged memory region, one per client connection, shared among the two callgates. This UID is set only after successful authentication by `login`. Thus, the only way for the `client handler` to read the e-mail of a specific user, is to log in as that user. If authentication is skipped, the UID will not be set, and the `e-mail retriever` will not fetch any mail. If the mail of a different user is being asked for, the `e-mail retriever` will not return it as it is programmed only to return e-mails belonging to the user set in UID. This architecture therefore satisfies our security goals.

One can reason about the security properties yielded by a partitioning by looking at diagrams, such as the one shown in Figure 3.2. If the `client handler` sthread is exploited, the attacker has no direct access to passwords or e-mails, since there are no direct arrows going from these memory regions to the sthread. The only privilege of the sthread (and attacker) is the ability to invoke the `login` and `e-mail retriever` callgates. To access passwords, the attacker's only option is to first exploit the `client handler` sthread, and then the `login` callgate. Note that the latter must be exploited via a very narrow interface—*e.g.,* a username and password. To lower the risk of exploits, the callgate can take care to ensure that input lengths and characters seem appropriate before processing them further. If the attacker manages to exploit the `e-mail retriever`, he will learn e-mails, though not passwords, and thus there are isolation benefits to multiple fine-grained callgates with different privileges. Finally, note that the partitioning shown in the diagram is instantiated once per network client (*i.e.,* one sthread and two callgates, per user). Thus, each network client will live in its own `client handler`, so there is no way for an exploited `client handler` to directly affect other users—there are no shared read / write memory regions across sthreads in the diagram.

To be certain that all of our security goals for our mail server have been met, we must assume that callgates are not exploited. For example, if the `login` callgate can be exploited to successfully authenticate without the correct password, then anyone's e-mail can be read. Similarly, if the `e-mail retriever` callgate can be exploited to retrieve an e-mail regardless of the setting of UID, then an attacker can read any mail. We must therefore implement these two components carefully.

We propose sthreads to protect memory regions and file descriptors. Our mechanisms are designed to prevent an sthread from reading an e-mail stored in another sthread's memory. It is worth noting that processes in today's OSes do not tackle this problem sufficiently. In UNIX, for example, process creation entails copying the parent's memory. This copying risks (potentially) leaking sensitive data, such as e-mails present in the parent process's address space, to the child process. An equally important goal is preventing sthreads from reading an e-mail directly from disk via system calls. We do not focus on this problem because previously proposed mechanisms, such as with SELinux, solve it. In fact, we leverage these solutions, and apply SELinux policies to each sthread to limit system call privileges. For example, even though in Figure 3.1 we show "passwords" and "e-mails" as being memory regions, in reality they are stored on disk and then loaded into memory as necessary. Using SELinux, we disallow the client handler from accessing these resources directly through the filesystem. System call protection, as provided by SELinux, is of critical importance to our whole security model to hold; protecting memory is of no use if an attacker could access resources directly from disk. The reverse argument holds too: what good is it to limit file system access if an attacker can read data directly from the process's memory? sthreads solve this latter problem.

3.8 Design patterns

How does one determine how an application should be partitioned? Getting the partitioning right is critical in assuring the security of an application, and it may seem like a daunting task. In our experience, though, we have noticed that types of applications tend to be partitioned similarly, according to patterns. We will now illustrate some of these patterns and provide guidelines as to how one may use sthreads.

In typical legacy network servers, a *master process* accepts connections. For such applications, client handling should then be done in a separate sthread for each client. This way, clients are isolated from one another, and the master is protected from its clients. Most programmers are already accustomed to this pattern, although they used `fork` or pthreads to deal with clients rather than sthreads. If the client sthread's permissions appear too great, such as including the ability to read a password database, then the programmer can introduce callgates, and give callgates these elevated privileges, while giving sthreads only the ability to invoke those callgates. In general, whenever there is sensitive data involved, a callgate is used to control access to it.

Client applications, or indeed all applications, are typically partitioned in a sim-

ilar fashion. Any "risky" operations, such as parsing, would be executed in an sthread with minimal privileges. If an sthread ever needs privilege, the privileged code executes in a callgate instead.

Because callgates need special attention for their security, we now discuss design rules to follow when implementing them:

- Minimize the extent of the untrusted arguments the callgate takes. A callgate should take simple-to-use arguments (such as integers) to limit avenues of exploitation. Taking in a structure containing many pointers or data that needs to be manipulated is far more risky. When possible, the sthread should parse data to provide callgates with ready-to-use data in order to avoid error-prone operations within the callgate.

- The output of the callgate must not be sensitive. Callgates will typically have access to sensitive data. Their output must in no way reveal information regarding this sensitive data. A subtle problem is that of callgates acting as oracles. Imagine a callgate implementing an encryption routine where the goal is to hide the encryption key. Although the callgate will successfully hide the key, the calling sthread will have full use of the key, since it can arbitrarily encrypt data, and hence use the key. The overall effect is a false sense of security, since protecting the key is no good if one can arbitrarily use it.

We believe that the above points are ever more important than reducing the size of code within a callgate to make auditing simpler. To illustrate this point, consider the extreme case of a very large amount of code within a callgate that takes no untrusted arguments. We argue that this callgate is difficult to attack because the caller has no direct inputs to leverage for exploiting the callgate. Conversely, a ten line callgate that takes a function pointer as an argument can become a real risk if that function pointer is followed without any checking.

3.9 Kernel implementation

We implemented sthreads for Linux as a 700-line kernel module and a 1,300-line userspace library. A few additional changes were required to the core kernel, for example, necessary to add new system calls and monitor the use of "dangerous" system calls that could affect the security of an sthread program. Programmers wishing to use sthreads are presented with the API shown in Table 3.1. The API is used as follows:

- To initialize sthreads, programmers must wrap their application's entry point

Sthread-related calls
`int sthread_create(sthread_t *thrd, sc_t *sc, cb_t cb, void *arg);`
`int sthread_exit(void *ret);`
`int sthread_join(sthread_t thrd, void **ret);`
`int smain(main_cb main, int argc, char **argv);`
Memory-related calls
`tag_t tag_new(int sz);`
`int tag_delete(tag_t);`
`void* smalloc(int sz, tag_t tag);`
`void sfree(void *x);`
`void smalloc_on(tag_t tag);`
`void smalloc_off();`
`BOUNDARY_VAR(def, id);`
`BOUNDARY_TAG(id);`
Policy-related calls
`void sc_mem_add(sc_t *sc, tag_t t, unsigned long prot);`
`void sc_fd_add(sc_t *sc, int fd, unsigned long prot);`
`void sc_sel_context(sc_t *sc, char *sid);`
Callgate-related calls
`void sc_cgate_add(sc_t *sc, cg_t cgate, sc_t *cgsc, void *targ);`
`void* cgate(cg_t cb, sc_t *perms, void *arg);`

Table 3.1: sthread API.

sthread security context (policy)	Description
`typedef struct sc {`	
`  unsigned long sc_mem[SC_MAX];`	Allowed memory tags and permissions
`  int           sc_memc;`	Number of allowed memory tags
`  unsigned int  sc_fd[SC_MAX];`	File descriptors and permissions
`  int           sc_fdc;`	Number of file descriptors
`  int           sc_uid;`	UID to run as
`  int           sc_gid;`	GID to run as
`  char          sc_chroot[PATH_MAX];`	Where to `chroot` to
`  char          sc_sid[256];`	SELinux SID (policy)
`  struct cgate  *sc_cgate;`	Allowed callgates
`} sc_t;`	
Callgate definition	Description
`struct cgate {`	
`  cg_t         cg_gate;`	Entrypoint
`  sc_t         cg_sc;`	Policy
`  void         *cg_targ;`	Trusted argument
`  struct cgate *cg_next;`	Next allowed callgate (linked list)
`};`	

Table 3.2: Data structures necessary for sthread creation.

with `smain`, which will invoke the real program entry point, specified in the argument `main`.

- The sthread creation and destruction APIs mimic those of pthreads, though the second argument of the create call is a security context (policy) rather than thread attributes.
- To tag memory, one first creates a tagged memory region with `tag_new`, specifying a maximum arena size, and then `smalloc`s memory indicating the tag, resulting in an allocation from the contiguous region for that tag.
- `smalloc_on/_off` are utility functions that will convert standard calls to `malloc` in the code executed between them to `smalloc`. These are useful when migrating existing code to sthreads, as the programmer may wrap a sequence of `malloc` calls without having to modify every single call.
- The `BOUNDARY` macros are used to tag globals and obtain their tag value at run-time, which is then used in policies.
- The functions prefixed with "`sc_`" are used to manipulate the security context of an sthread or callgate. They allow adding memory permissions, file descriptors, setting of an SELinux policy, and adding callgates.

- A callgate is invoked using the `cgate` function.

To create an sthread, one must enumerate its privileges using the data structures shown in Table 3.2. sthread creation will setup existing kernel data structures like page tables and file descriptor tables, which are then used for privilege checking at runtime. At creation, tagged memory regions are referred to by their base pointer, and permissions are encoded in lower bits. Note that tagged memory regions are page aligned, with zeroed lower bits, leaving enough space to encode combinations of read / write / COW in lower bits. File descriptors and their permissions are encoded in a similar fashion too, though the file descriptor number is shifted three bits left to reserve space for permission bits. Information about allowed callgates is stored in a linked list so that callgates can later be instantiated at sthread creation time.

We now discuss in detail the implementation of sthread initialization, sthread creation, tagged memory and callgates. We note that all of sthreads are implemented in userspace, apart from "sthread recycling", a concept that we shall later introduce for performance reasons.

3.9.1 Sthread initialization

The role of `smain` is to initialize sthreads. It will create a copy of the process' memory so that all new sthreads can be created with that pristine memory by default. All sthreads will therefore inherit code, data, heap and stack available at the time `smain` is called. Because this is called at the very beginning of the program's execution, it contains no sensitive information that is yet to be created by the application. In our current implementation, we do not consider program argument values and environment variables to be sensitive (present in the stack). To simplify our explanation (and implementation) we shall discuss statically linked applications only. Our implementation supports dynamically linked binaries too, though it requires more bookkeeping than explained for static binaries, as more memory regions are present in dynamically linked binaries.

When a statically link binary on Linux is executed, its memory map is as follows:

- Read-execute text. (.text and .rodata.)
- Read-write data. (.data and .bss.)
- Read-write heap.
- Read-write stack.

When `smain` is called from `main`, `smain` enumerates all the read-write segments by reading /proc/self/maps, and does the following for each read-write memory segment:

1. Creates a new POSIX shared memory segment using `shm_open`.
2. Copies, with `memcpy`, the contents of the read-write memory segment into the new POSIX shared memory segment.
3. `mmap`s copy-on-write (COW) the POSIX shared memory segment over the original read-write memory segment. This simplifies the implementation as we can undo any memory changes by simply remapping the segment. It is perfectly acceptable for the parent sthread to manually track memory changes and undo them when creating a child, without sacrificing security.

The POSIX shared memory segments consist of the original memory contents of all the read-write memory segments present at the time `smain` was called. In other words, we have a copy of initial data, heap and stack. We also keep track of the virtual addresses and length of each of those segments, so that we can restore them at sthread creation time. Dynamically linked binaries would have multiple data segments, one for each library, requiring more bookkeeping. When we create a new sthread, we can now restore pristine memory contents by doing a `mmap` COW for each memory segment. This way, any content modifications to these segments made by the parent are not leaked to the child. Also, COW prevents any writes performed by the child sthread from corrupting the pristine memory segment. Segments are restored with their original initialized values rather than zeros, so that any library code that expects initialized structures, such as `malloc`, works. Indeed we call `smain` after `main` to allow for such initializations to occur, rather than having to reinitialize on each sthread creation.

3.9.2 Sthread creation

To create an sthread, we do the following:

1. Create a new process using `fork`.
2. Setup initial memory according to the security context specified as parameter to `sthread_create`, and to the initial state as per `smain`. That is, for each memory segment of the process (as per /proc/self/maps):
    - If the segment was present at the time of `smain`, we `mmap` it COW back to its original copy, stored in a POSIX shared memory segment. Note that if the segment expanded (*e.g.,* heap), we `munmap` any extra length.

- If the segment is present in the security context, we `mprotect` it according to the permissions in the security context. If the segment is passed as COW, we create a copy of it now for ease of implementation, hence manually unsharing pages and increasing memory usage.
- Otherwise, if the segment is not allowed by the security context, and was not present at `smain` time, we `munmap` it.

3. Setup file descriptors according to the security context. For each file descriptor (as per /proc/self/fd/), we:
   - If not present in the security context, `close` it.
   - Otherwise, `fnctl` or `shutdown` as appropriate to set permissions according to the security context.
4. `chroot` to the directory specified in the security context.
5. Drop privileges with `setuid` and `setgid` to the user/group specified in the security context, or the default sthread ones.
6. SELinux transition to the policy specified in the security context, or the default-deny sthread policy.
7. Execute the sthread's callback.

Note that dropping privileges prevents sthreads from tampering with the POSIX shared memory segments created at `smain`, or created for tagged memory regions. These are created by the master sthread, running as root, and hence owned by root. Our current implementation requires the master sthread to run as root in order to `chroot` and `setuid` child sthreads. While this is a limitation, we do not see it as a major one, especially because the privileges of root can be controlled using SELinux. Despite sthreads running with the same userid, they cannot `ptrace` or send signals to each other, as mandated by the SELinux policy. Access to `/proc` can be denied too with SELinux or filesystem permissions.

We note that there is much overhead associated with creating an sthread. This is partly due to the effects of `fork`, and partly due to setting up appropriate memory maps. We analyze this cost in detail in Chapter 6. To mitigate this cost, we introduce the concept of sthread recycling. It is a very common design pattern to create sthreads that have the same permissions, so we attempt to optimize this case. For example, when serving a network client, we might create an sthread that has access to some configuration data, and a tagged memory region containing arguments. When this sthread exists, we may need to create an identical one to serve

Sthread-related calls
`int checkpoint();`
`int restore(sthread_t thrd, int *new_fds);`

Table 3.3: sthread recycling system calls.

the next client. Rather than creating and destroying sthreads, we keep a pool of long-lived sthreads, a standard technique used to improve performance. To preserve the security semantics of `sthread_create` though, we need to ensure that memory and other sthread aspects are properly reset prior to reuse. We call this sthread recycling.

3.9.3 Sthread recycling

To support sthread recycling, we add two kernel APIs, shown in Table 3.3. When an sthread is first created, before its callback is invoked, it is in a pristine state. It does not yet contain sensitive data generated by the sthread, nor it is yet exploited. We therefore `checkpoint` this sthread to preserve this state. Once the sthread terminates, to create a new one, we `restore`, rather than going through through the slow-path via `fork`. The semantics of `checkpoint`, upon `restore`, are:

- The memory map of checkpointed memory regions cannot change. That is, if one attempts to `mmap`, `brk`, `munmap`, `ipc`, `mprotect`, `mremap`, `remap_file_pages` in any way that affects checkpointed memory regions, then `restore` will fail. `execve` too will cause `restore` to fail. Note that we still allow these operations as they may be legitimate, *e.g.,* `execve` for CGI programs. We fail `restore` to indicate that the sthread is unrecyclable, and that a new one needs to be created via `fork`.

- Any checkpointed COW memory is restored to its original content. It is a requirement that COW memory is backed by a POSIX shared memory segment, holding original contents.

- Checkpointed file descriptors may not change. That is, one cannot `close`, `ioctl`, `fnctl` such file descriptors, or `restore` will fail. Note that we allow changing the file (seek) pointer as its "default" position is application dependent. Applications are responsible for resetting this.

- Any new memory, *i.e.,* memory not present at time of `checkpoint`, is unmapped. The same occurs for file descriptors, *i.e.,* new file descriptors are closed.

- Any other process state is restored. Specifically, signal handlers, process limits, and the working directory.

Note that some file descriptors are "volatile" and change between sthread runs. Examples are sockets. Socket number five could represent one user in this run, but a different user on the next sthread creation. Thus, `restore` allows sending new file descriptors to the recycled sthread. We do not use UNIX sockets to pass file descriptors as we found this mechanism to be slow, as we show in Chapter 6.

To implement `checkpoint`, we do the following:

- The process is marked as being an sthread. This enables code in system calls such as `mmap` to check whether checkpointed memory regions are being changed in an sthread, setting a flag that will cause `restore` to fail, if necessary.
- The original memory map is copied.
- The file descriptor table is copied.
- Other process state such as signal handlers and process limits are copied.

To implement `restore`, we do the following:

- We check whether the flag to fail `restore` is set. This is set, for example, if a process attempts to change the checkpointed memory map. If the flag is set, we fail `restore`, requiring sthread creation via `fork`.
- We unmap any new memory that was not present at the time of `checkpoint`. The only exception is the stack, which we leave mapped but proactively zero it. We found this to perform better than removing new stack memory, and taking page-faults to dynamically grow it again on the next run.
- For all read-only memory, we do nothing. Exploits targeting such memory are caught proactively by the restore flag, directly in system calls such as `mmap`.
- For read-write (non-COW, *i.e.,* shared) memory, we do nothing. The only semantics are to ensure that such memory points to the checkpointed memory pages, and this is checked proactively in calls like `mmap`. Such memory is typically used for return values of an sthread, and the kernel does not scrub it because it does not know whether the return values have yet been processed and are ready to be reclaimed. This memory is instead scrubbed by our userspace component and will be detailed in Section 3.9.6.

- For COW memory, we walk the page table to determine which parts of the COW mapping have been written to. We restore such pages with their original contents, using `memcpy`. We found this to perform better than reclaiming these pages and taking page-faults in the future.

- We close any new file descriptors not present at the time of `checkpoint`. We install any new file descriptors passed as an argument to `restore`.

- We restore saved process state such as signal handlers, brk limit, process limits and the working directory.

- We restore registers, which also causes the sthread to start executing.

sthread creation via `fork` or `restore` is handled by our library. The end-user API remains `sthread_create` and the user need not be aware of sthread recycling. To recycle sthreads, our library must know which sthreads have completed execution and what their privileges were, so that our library can find a candidate sthread for recycling. If a dormant sthread with the same privileges of the sthread that is being created is found, then recycling occurs. sthread privileges must be the same so that the memory maps of the new and old sthreads equal, a requirement for recycling using `checkpoint` and `restore`. The bookkeeping done by our userspace library for recycling is:

- On child sthread creation via `fork`, create a data structure holding a copy of the sthread's security context and sthread ID, storing it in a linked list in the parent's memory. Mark the newly created sthread as running in its userspace data structure.

- When the sthread is joined, mark the sthreads as recyclable in its userspace data structure. Note that the sthread does not `exit` in the kernel, but rather recycles itself and goes to sleep, so that it can be reused in future.

If a new sthread needs to be created, first, our library traverses the linked list of running/recyclable sthreads to find a recyclable sthread with the same privileges as the one that is about to be created. If one is found, `restore` is called upon it. Otherwise, the sthread is created fresh using `fork`.

Finally, `checkpoint` and `restore` is the only extra functionality we require from the Linux kernel to implement sthreads efficiently. Note that the end-user API remains unchanged, though the sthread userspace library uses this new kernel functionality to improve performance where possible, as shown in Figure 3.3.

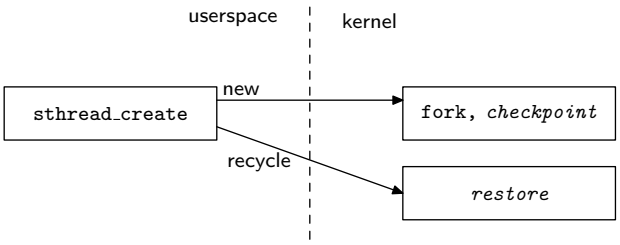


Figure 3.3: Relationship between the sthread userspace API and kernel API. In italics, we depict new kernel functionality required for sthreads.

3.9.4 System call protection

sthreads only protect memory and file descriptors. We use SELinux to control system calls. We need system call protection for two distinct reasons:

1. Control interactions between sthreads and the rest of the system. For example, we might want to limit filesystem and network access. We leverage SELinux for this control.

2. Preserve invariants between sthread recycles. When we recycle an sthread, we assume that some process state does not change. For example, we assume that `checkpoint`ed memory regions are not altered via `mprotect`. The only way to alter checkpointed invariants is via system calls, so we monitor system calls to detect any changes, and fail `restore` if needed. To control these aspects, we add kernel checks to system calls that toggle a "fail restore" tag, per sthread, if checkpoint invariants change.

Table 3.4 lists all the system calls in Linux 2.6.28. For each system call, we briefly state how we control it, and whether it is important for sthread recycling.

Because sthreads run with the same userid, care should be taken when creating resources that could be opened by other sthreads (*e.g.,* temporary files). By default, our SELinux policy prevents the creation and opening of such objects. Thus, a programmer has to explicitly open a door. To avoid errors, an alternative approach is to run each sthread with a different ID, which is already possible with the current API. We chose not to do this by default merely to avoid forcing system administrators to allocate user IDs specifically for sthreads, especially when considering that some applications (like Apache under load) may require thousands of sthreads (hence UIDs). Other techniques like `chroot` or `jail` in a per-sthread location can also be used. In the current implementation, to share a file between two specific sthreads, one can create an SELinux domain that has exclusive access to a part of the filesystem, and run only those two sthreads in that domain. An alternative is to

create a callgate that is programmed to access only a particular file, and only give two particular sthreads access to that callgate. The same can be done for sharing other resources, or controlling how sthreads signal each other.

System calls	Control action
exit read write waitpid time seek getpid pause access times getuid getgid getppid getpgrp sgetmask sigsuspend sigpending getrusage gettimeofday getgroups select poll getpriority getitimer wait sysinfo sigreturn getpgid msync readv writev getsid fdatasync sched_getparam sched_getscheduler sched_yield sched_get_priority_max sched_get_priority_min sched_rr_get_interval nanosleep getcwd capget sendfile getrlimit mincore gettid readahead futex sched_getaffinity get_thread_area io_destroy io_getevents io_submit io_cancel exit_group epoll_ctl epoll_wait set_tid_address timer_settime timer_gettime timer_getoverrun timer_delete clock_gettime clock_getres clock_nanosleep get_mempolicy mq_unlink mq_timedsend mq_timedreceive mq_notify mq_getsetattr waitid ioprio_get inotify_add_watch inotify_rm_watch get_robust_list splice sync_file_range tee vmsplice getcpu	Allowed.

System calls	Control action
`pipe dup epoll_create inotify_init timerfd_create eventfd`	Allowed. New FDs are closed on `restore`.
`close dup2`	Allowed. Closing a non-volatile (*e.g.,* non-socket) checkpointed FD will fail `restore`.
`fork vfork clone`	Allowed, but all children must have `exit`ed on `restore`.
`execve uselib mmap brk munmap mprotect mremap remap_file_pages`	Allowed. If changes are done to `checkpointed` memory map, `restore` fails. (`execve` always fails `restore`.) New memory is unmapped.
`chdir signal ioctl fcntl umask sigaction ssetmask sigprocmask flock mlock munlock mlockall munlockall sigaltstack`	Allowed. Checkpointed values are reset on `restore`.
`alarm setitimer`	Allowed. Any timers are cleared at `restore`.
`io_setup timer_create mq_open`	Allowed. Must be followed by an appropriate `destroy` before `restore`, or recycling will fail.
`link unlink mknod chmod chown stat mount stime utime sync rename mkdir rmdir umount uname chroot ustat statfs sethostname settimeofday symlink readlink swapon swapoff reboot readdir getdents truncate syslog setdomainname adjtimex init_module delete_module quotactl bdflush sysfs sysctl nfsservctl pivot_root setxattr getxattr listxattr removexattr clock_settime kexec_load`	Controlled by SELinux.

System calls	Control action
`ptrace kill tkill`	Controlled by SELinux, but dangerous for sthreads. By default, these are blocked by the default-deny policy. Enabling them would likely break sthreads. Callgates should be used to limit the use of these system calls.
`open creat socketcall ipc`	Controlled by SELinux. Any new FDs will be closed on `restore`.
`setuid setgid nice acct setpgid setsid setrlimit setgroups setpriority ioperm iopl vhangup vm86 modify_ldt personality setfsuid setfsgid sched_setparam sched_setscheduler prctl capset madvise sched_setaffinity set_thread_area mbind set_mempolicy add_key request_key keyctl ioprio_set migrate_pages unshare set_robust_list move_pages signalfd`	Denied. These affect recycling, and we expect them to not be used in sthreads as many such calls are privileged. The master (first) sthread can invoke them though. Many of these calls setup execution parameters for the sthread, so if necessary, the master can set these up for the sthread, before dropping privileges and starting it. It will not be possible to change the parameters between recycles, though.

Table 3.4: System calls in Linux 2.6.28 and how they are controlled for sthreads.

3.9.5 Callgates

Callgates can be built entirely from sthreads. To create a callgate, we do the following:

- Create a tagged memory region for callgate arguments and invocation semaphores.
- Create an sthread, for the callgate, that has access to its tagged memory region for arguments.
- The callgate's sthread waits on a semaphore that is used to signal invocation.

To grant callgate access to a child sthread, we merely allow the child to access the tagged memory region associated with the callgate. This allows it to pass arguments to the callgate, read return values, and invoke the callgate. To invoke the callgate we signal a semaphore. To wait for callgate completion, we wait on another semaphore.

All of this is hidden under the callgate API. Note that the underlying mechanism supports for asynchronous callgate invocations, if needed.

To improve performance but sacrificing some security, one can reuse callgates. Instead of creating a new callgate each time, one can have a single long-lived callgate that is reused upon each invocation. This is different from *recycled* sthreads, as with callgates, we simply *reuse* them as-is. Because we assume that callgates remain unexploited, we need not scrub their memory or "recycle" them before reuse. On the other hand, if a callgate does get exploited, reusing callgates between sthreads means that an attacker has a means for spreading to other sthreads, via the shared callgate.

3.9.6 Tagged memory

We use hardware page protection to enforce memory privileges. As a consequence, we need to place data with different tags in different pages, so we need to control memory allocations. To implement tagged memory, we simply use POSIX shared memory. We create a new tagged memory region with `shm_open` and `mmap`. To implement `smalloc`, we wrap `malloc` instructing it to allocate out of the arena specified by the tag parameter in `smalloc`. To implement `smalloc_on` and `smalloc_off`, that convert standard `malloc` calls between them to `smalloc`, we use malloc hooks. To tag globals, we declare them with the BOUNDARY macro which places variables in a page-aligned ELF section. This way, globals with different tags are placed in different pages when the binary is loaded. We then "convert" globals to standard tagged memory regions by explicitly creating a POSIX shared memory segment, copying the values of globals, and `mmap`ing the shared memory segment over the globals.

We optimize the creation of tagged memory regions by keeping a free list of arenas in userspace. Rather than deleting tags by calling `munmap`, we scrub (`memset` zero) tags and keep a free list in userspace. When a tagged memory region is next needed, we return it from our free list, if available. This avoids the cost of `mmap`. Recall that our `restore` implementation does not scrub read-write (shared) memory regions. These are typically used for return values from child sthreads. Thus, when a parent finishes processing a return value from a child, it will delete the tag. This will cause the tag to be scrubbed, hence providing secrecy.

3.10 Userspace implementation

In our kernel implementation, we implemented most sthread functionality in a userspace library, factoring out the `checkpoint` and `restore` functionality into the

kernel. Is it possible to implement `checkpoint` and `restore` in userspace, too, and thus to deploy sthreads on legacy UNIX systems? We show that it is, though at some performance cost. Our userspace implementation consists of 5,000 lines of code. Although we only tested it on Linux, we use standard POSIX and System V calls, and avoid Linux-specific calls, so our general design should work on other UNIX systems, too.

The main challenge when implementing sthreads in userspace is: how does the parent force a child to recycle? In our kernel implementation, the kernel cleans up the child and the parent trusts the kernel to do so. In a userspace implementation, the child could have a routine that recycles the process, but how does the parent guarantee that the child ran it and is not lying that it did? A malicious child could tell the parent that it is clean, whereas an exploit is still resident in memory, and the parent would then reuse this polluted child. If this child were then responsible for serving a new network client, it would have access to any sensitive information the client gives out.

Note that a parent cannot directly clean up a child because a process cannot manipulate the memory of another process—after all, UNIX processes are meant to restrict this to provide isolation. Of course, in our kernel implementation, our kernel code was free to manipulate process memory at will. Since the child must perform the cleanup, there are two sub-problems: how does the parent ensure that the child still has the correct clean-up routine, and how does it force the child to run it?

To solve the first problem, we use System V shared memory. For each shared memory segment, the kernel keeps statistics such as the last PID that attached, and most importantly, the last detach time. These are retrievable, using `shmctl`, by any process that is allowed to read the segment. If a process attempts to unmap a System V shared memory segment, the segment's last detach time will be updated. We use this to detect exploits as follows. We create a new System V shared memory segment and copy all of our `.text` segment in it. We then remap our `.text` segment, pointing it to this new System V shared memory segment. In other words, the process now executes reading its code from a System V shared memory segment rather than a memory mapped file on disk. When forking, children will inherit this segment, and the attach time will be updated, but the detach time not. If a child ever unmaps that segment (*i.e.,* replaces its code), then the detach time of the segment will be updated, and the parent can check that. Thus, the parent has a mechanism for checking whether code, and hence a restore routine, is still present in the child, at the expected memory address.

The next problem is: how do we force the child to run the restore routine? To do this we use `ptrace`, a system call used by debuggers to inspect (and alter) memory

and registers of other processes. The parent can change the program counter (EIP) of the child to point to the restore routine. The parent is now certain that the child is executing the expected code. Note that we continue to `ptrace` the child at all times in order to intercept signals. For example, a malicious child could attempt to subvert execution by setting an alarm that would invoke a signal handler sometime in the future, potentially stopping the restore routine. With `ptrace`, we are notified of all signals and can stop (or allow) them as necessary. To prevent races between the time we check the existence of the restore routine and its execution, the parent stops the child using a signal. The restore routine then executes as follows, in a similar fashion as the kernel implementation:

1. For all read-only (and execute) segments, we can verify that they still exist and point to the right segment using the detach time in the System V shared memory statistics.

2. For shared write (non-COW) segments we can do the same, using System V statistics.

3. For COWed write segments, we need to remap them COW using `mmap`. Note that `mmap` requires a file descriptor, and we need to verify this using `fcntl`, making sure that it still points to the expected device and i-node. We use this technique to verify other file descriptors, too.

4. For "volatile" file descriptors, *i.e.,* those that change across different sthread executions (*e.g.,* sockets), we must obtain new copies via a UNIX socket. Note that in the kernel we can update file descriptors using the argument to `restore`. In userspace, the only mechanism we have for doing this is UNIX sockets, which are considerably slower.

We now discuss one last detail of our userspace implementation that affects its performance in certain cases. Child sthreads incur an additional overhead when creating their own child sthreads and tagged memory regions. When the master (first) sthread does so, there is no such overhead, though, so this only affects sthreads lower in the family tree. In our implementation, the master sthread is required to run as root and all child sthreads run as non-root. This is required to enforce memory protection. When creating a tagged memory region, we use either System V shared memory or POSIX shared memory, and in both cases, the creator of the region can specify who can attach to these regions. The permissions are standard UNIX permissions relating to owner, group, and others. We create all segments owned by root with mode 0600, *i.e.,* only root can read / write (and hence attach)

to them. By running all child sthreads as non-root, we restrict sthreads from attaching to arbitrary tagged memory regions and hence are able to enforce sthread policies. When creating an sthread, it first runs as root, and we are therefore able to attach the relevant tagged memory regions as specified in the security context for the sthread. We then drop root privileges and invoke the sthread callback, *i.e.,* the actual application code for that sthread. Having all memory segments owned by root has implications as to who can create them and attach to them. The master sthread, running as root, has no restrictions and can do all of these operations directly. Child sthreads, though, which run as non-root, will need to contact the master in order for it to create new tags and sthreads on the child's behalf. Hence, for applications where multiple levels of sthreads are involved, our implementation is less fast because of the extra IPC and management required to handle these cases.

3.11 Security analysis

We now attempt to attack our sthread implementation to assess its robustness. We categorize attacks as follows:

1. Disclosure of sensitive memory and file descriptors. We examine the memory and file descriptors sthreads inherit by default, and check whether any sensitive information can be obtained.

2. IPC with other sthreads. We examine whether an sthread can disrupt other sthreads, via, for example, signals.

3. Filesystem interactions. We examine possible attacks that can occur via the filesystem.

4. Other system calls. We examine avenues of attack via system calls.

5. Defeating sthread recycling. We examine whether an attacker can influence future sthread uses, after recycling.

6. Denial of service attacks. We examine how an attacker can consume resources on the host.

Our goal is to prevent attackers from learning sensitive information, prevent attackers from affecting other sthreads apart from those exploited, and to preserve sthread recycling semantics. We do not tackle denial of service attacks.

Disclosure of sensitive memory and file descriptors. We create a new sthread and the attack consists of reading all available memory and from all file descriptors. Our sthread implementation, by default, creates new sthreads with no file descriptors. There is no shared memory with the parent, so the parent's integrity is preserved. Some memory though is inherited by the child. Specifically, this is the sthread's memory at the time `main` was called. We note that this contains arguments to the program and the system's environment variables. This is the most prominent information leak in our implementation. It is not fundamental—when checkpointing at `main`, one can zero the top stack pages to prevent child sthreads from learning arguments and environment. Another possible information leak is any sensitive data computed by constructors (most likely in C++ programs), as these execute before `main` is called. Thus, our sthread implementation currently does protect the privacy of program arguments, system environment variables, any data computed by constructors, and any other statically initialized data. We do however preserve the secrecy of the parent's dynamic memory and file descriptors.

IPC with other sthreads. We perform two attacks: `ptrace` and `kill`. Note that these are the only two IPC calls where the attacker does not need the "consent" of the victim to perform IPC. Other IPC requires collaboration of both parties involved in the communication, *e.g.,* setting up sockets and explicitly accepting connections, so the victim must explicitly open doors for attack. The aim of the `ptrace` attack is to attach to an sthread and read / write its memory and registers. The aim of the `kill` attack is to attempt to corrupt the state of an sthread with vulnerable signal handlers, or to kill an sthread (DoS attack). Because all sthreads run with the same UID, they are able to `kill` and `ptrace` each other, according to standard UNIX permission checks. In our kernel implementation, both these attacks failed due to our default-deny SELinux policy. One would need to explicitly allow `ptrace` and individual signals via the SELinux policy. We do not foresee the use of `ptrace` in production runs to be necessary as it is mainly used for debugging (or hacking). To enable sending signals among cooperating sthreads, one must use callgates. SELinux alone is insufficient as it is likely that all sthreads will run in the same domain, making it impossible to specify at fine granularity which sthread is allowed to send a signal to which other sthread. Callgates, instead, could run at a privileged domain that allow sending signals. One can therefore create a callgate that sends a signal to a specific sthread, and allow other sthreads to invoke that specific callgate, thereby controlling signal interactions among sthreads.

Our userland implementation denies these attacks without using SELinux. A process can only be `ptrace`d by one process. Thus, the master `ptrace`s all sthreads,

preventing sthreads from using `ptrace` on each other, as UNIX allows only one tracer per process and each sthread is already traced by the master. `ptrace` also routes all signals to traced processes to the tracer first, for management. Thus, the master can monitor all signals sent to sthreads, and drop them if necessary. This stops the `kill` attack.

Filesystem interactions. We attempt three attacks: accessing regular files, accessing /proc, and accessing POSIX shared memory files (used to implement tagged memory regions). sthreads, by default, all run with the same userid, so they can all read each other's files. If one sthread writes sensitive data to disk, other sthreads can read it. Unfortunately it is up to the programmer to avoid such risks. For example, if implementing an FTP server, the master sthread should create sthreads with the userid of the user that logged in rather than the generic sthread user, to avoid other sthreads (*e.g.,* pre-authentication) from reading uploaded files. Alternatively, one can prevent sthreads from writing data by using a stringent SELinux policy that does not allow disk writes (this is the default). If an sthread needs to write temporary data, the master should `chroot` the sthread in a private, per-sthread, directory. Note that access to other system files can be limited by using standard UNIX permissions or SELinux.

The next attack involves accessing /proc. On Linux, /proc allows, among other things, to read/write the memory of other processes running with the same userid as the process accessing /proc. Because all sthreads run with the same userid by default, they could access each other's memory via /proc according to standard UNIX permissions. The attack fails due to our default-deny SELinux policy. Note that in SELinux's default policy, all domains are able to access their own /proc entry. Thus, we had to relabel the /proc directory, allowing all domains except the default sthread one from accessing /proc. An alternative, which our userspace implementation uses, is to `chroot` sthreads. If `chroot` is undesirable, one can change the UNIX permissions of /proc, allowing no world access, but standard group access for users (but not sthreads).

Our last attack in this category involves accessing tagged memory regions via the filesystem directly. Because we use POSIX shared memory for tagged memory regions, they are visible in /dev/shm. Since all sthreads run with the same userid, if an sthread creates a new tagged memory region, any other sthread can attach to it (if not using `chroot`). To avoid this, our current implementation creates the segment and immediately `unlink`s the file, so nobody can attach to it in the future. Admittedly, there is a race condition, even though the attacker needs to guess (we make /dev/shm non-readable) a random name in a very short window of time. A

more mature implementation would create (and remove) the shared memory segment atomically, in the kernel. Alternatively, one could allow only `creat` but not `open` in /dev/shm, using SELinux.

Preventing this attack in our userspace implementation is more complex. First of all, running each sthread in a different `chroot` is insufficient. In our userspace implementation, we also use System V shared memory, which is not associated to the filesystem. The only protection there, are userids. We therefore cause the master (running as root) to create all tagged memory regions. sthreads, running as non-root, can no longer attach to arbitrary segments. When an sthread needs to create a new tagged memory region, it contacts the master to create it on the sthread's behalf. In order for the child sthread to attach to the new segment, the master will need to lower the segment's privileges temporarily. If an attacker sthread attempts to attach to the segment, the master can detect this by inspecting the segment's statistics, which reports the number of processes currently attached to this segment. For POSIX shared memory segments, the master sends the segment's file descriptor to the legitimate sthread via UNIX sockets. The master knows that the UNIX socket points to the correct sthread because UNIX sockets allow passing credentials with messages, which include the PID of the sender. Child sthreads cannot open the file descriptor of the POSIX shared memory segment themselves, because it is accessible only to root, as it is created by the master sthread.

Other system calls. Sthreads run as non-root, so dangerous system calls like `reboot` or `setuid` will fail. SELinux allows control over remaining dangerous system calls. If SELinux is unavailable, firewall rules can be used to limit network traffic of sthreads, by, for example, matching particular UIDs or PIDs. Unfortunately, SELinux does not restrict some system calls like `gettimeofday` so the attacker may be able to disclose certain system statistics or information. We have not found mechanisms for actively attacking other sthreads or leaking sensitive information from them. Finally, kernel exploits will succeed unless SELinux blocks vulnerable system calls.

Defeating sthread recycling. So far we assessed, among other things, whether one sthread can affect another one. Because we recycle sthreads, can a past sthread invocation affect a future one? That is, are the security semantics of recycling an sthread or creating one anew, equal? We attempt three types of attacks: altering the memory map, altering file descriptors, and altering other process state. To alter the memory map, we attempt both modifying existing mappings, and adding mappings. Modifying existing mappings with `execve`, `brk`, `mmap`, `munmap`, `ipc`,

`mprotect`, `mremap`, `remap_file_pages` all set a flag that will cause future kernel `restore` calls to fail. Note that `brk` will set the flag only if the limit is reduced—if growing heap increases the limit, it will be reset at `restore`, without failure. In our userspace implementation, all these calls are detected by a change in the detach time in System V statistics, except for `mprotect`. The attacker can `mprotect` a mapping from read-execute to read-only, so we must proactively `mprotect` back to read-execute (or simply attempt to execute). Note, an `mprotect` from read-only to read-write fails, as these permissions are enforced by shared memory segments (used throughout). All new memory mappings have been removed at `restore` in our kernel implementation. In our userspace implementation, we rely on the sthread doing appropriate cleanup. A malicious sthread not cleaning up can be detected by checking memory usage via `getrusage`.

Our next attack involves altering file descriptors. Calling `close`, `ioctl`, `fnctl`, `flock` on existing file descriptors all set a flag that will cause `restore` to fail. We do not reset the file pointer, so it is the sthread's responsibility to reset this, if needed. (We do not do this because it depends on the application's semantics whether the file pointer should be set to the beginning or end of file, or elsewhere.) Our userspace implementation ensures the validity of the file descriptors via `fstat` and resets flags via `fcntl`. Sockets cannot be verified via `fstat` though these are typically "volatile" and resent (hence reset) when recycling. Any new file descriptors are closed by our kernel implementation. Our userspace implementation is more relaxed and leaves any "leaked" file descriptors open.

Our final attack in this category involves changing other process state. We attempt to change signal handlers. The kernel implementation restores these. Our userspace implementation instead, for efficiency, masks all signals (a single system call). Thus, it is the sthread's responsibility to reinstall any signal handlers used. Any leftover timers and alarms set are removed in our kernel implementations, and their notifications (signals) masked in our userspace one. We explicitly reset the sthread's working directory with `chdir`. Changing other process state like limits (`setrlimit`) fails as per our default-deny SELinux policy. Unfortunately, `setrlimit` succeeds in our userspace implementation, allowing a DoS attack, by, for example, reducing the limit of maximum memory allocation. A workaround is to pre-allocate enough memory when checkpointing at `main` so that all sthreads have at least that much memory. Finally, we detect whether an sthread still has active children. This is trivially done in the kernel by checking whether the process control block has children. In our userspace implementation the master can detect `fork` (and thread creation) as it is notified via `ptrace`.

Denial of service attacks. We attempt a `fork` bomb: *i.e.,* creating processes as fast as possible. It succeeds in our current implementation, essentially making the machine extremely slow. This can be mitigated by using `setrlimit`, or detecting `fork` calls (possible in userspace too, using `ptrace`). We attempt to exhaust memory, and this also succeeds. Again, we can limit memory usage with `setrlimit`. Some process limits are per-userid so care must be taken when wishing to apply them per-sthread as by default sthreads run with the same userid. Finally, we do not handle `nice` and changing a process' priority in our userspace implementation. A more mature implementation could have the master `renice` child sthreads upon recycle. The use of `nice` can be controlled via SELinux in our kernel implementation.

3.12 Limitations

We now review the limitations of sthreads, and topics that bear further investigation. Several of these limitations concern callgates. First, we rely on callgates not being exploitable. Second, the interface to a callgate must not leak sensitive data: neither through its return value, nor through any side channel. If a return value from a callgate does so, then the protection within the callgate is of little benefit. More subtly, callgates that return values derived from sensitive data should be designed with great care, as they may be used by an adversary who can exploit an unprivileged caller of the callgate either to derive the sensitive data, or as an oracle, to compute using the sensitive data without being able to read it directly.

We trust the kernel support code for sthreads and callgates. As this code is of manageable size—fewer than 2,000 lines (5,000 for our userspace implementation)—we believe that it can be audited, and it only need be audited once, to be usable with many applications. Perhaps more worryingly, we must also trust the remainder of the Linux kernel, though like Flume [34], we also inherit Linux's support for evolving hardware platforms and compatibility with legacy application code.

Sthreads do not deny read access to the text segment; thus, a programmer cannot use the current sthreads implementation to prevent the *code* for an sthread (or indeed, its ancestors) from being disclosed. Sthreads provide no direct mechanism to prevent DoS attacks, either; an exploited sthread may maliciously consume CPU and memory. Sthreads will also leak memory addresses which are sensitive if address space layout randomization [70] techniques are used to prevent exploits. This information could be used to attack callgates, although it is certainly possible to construct the system in such a way that callgates and sthreads have different memory layouts, thus eliminating this particular attack.

3.13 Summary

Sthreads allow the splitting of applications into reduced privilege compartments. Doing so minimizes the impact of exploits, since they will be contained in the exploited compartment, which typically runs with reduced privilege. Sthreads consist of three main abstractions. First, sthreads themselves define unprivileged compartments that receive no privileges by default. An sthread must be granted explicit privileges to access memory and file descriptors, and include an SELinux policy in order to control its interactions with the operating system. The second abstraction is tagged memory, which provides a mechanism for grouping objects into memory regions and enforcing memory permissions across sthreads. The third abstraction is a callgate, which is a privileged sthread. Each sthread can be granted the ability to invoke a callgate to perform privileged operations on its behalf. The attack surface of an application is therefore restricted to callgates, which are typically much smaller than sthreads as they perform small and specific tasks. The intended use of sthreads is to encapsulate all risky code (*e.g.,* user input parsing) into sthreads and protect sensitive data behind callgates.

We have both a kernel and userspace implementation of sthreads, for Linux. Our kernel implementation shows that sthreads require only a small extension to commodity OSes (≈2,000 lines of code). Our userspace implementation shows that even without new kernel functionality from Linux, one can still implement sthreads, though at some performance cost, as we show in Chapter 6. To improve performance, we introduce "sthread recycling", which allows safely reusing sthreads from a pool, rather than creating new ones each time. In Chapter 5 we show that these relatively simple mechanisms are sufficient to secure a wide range of applications.

Chapter 4

Tools for securing legacy applications

There has been significant work on how to write secure partitioned applications. Our sthread mechanism is one such proposal, though it focuses on extending existing concepts, modifying them for better isolation of sensitive data. However, there has been little work on how to partition existing code. It is clear how one may write partitioned applications from scratch, if one were to design them with partitioning in mind. If one instead were writing a standard non-partitioned application, one would typically leverage the fact that all memory is available to all code, and the resulting implementation would therefore have quite intricate memory dependencies, making the whole application rather monolithic. Much existing code indeed is written in such a way, and the question remains how to split it. One approach would be to study the code carefully and figure out how to pull it apart, much like the authors of privilege-separated OpenSSH [57] did. Their work has in fact shown that such an approach is rather difficult. Manually studying the code and partitioning it becomes much more tedious as the complexity of the application grows. OpenSSH is a relatively simple application, for which manual partitioning is tractable, although this may not be the case for all applications, such as complex databases. We now address the problem of how to ease the partitioning of existing applications. Our solution is to provide a set of tools that minimizes the amount of code one must study.

Applying sthreads to existing code is tedious because the programmer must somehow determine the permissions of the compartments he "carves out." Any section of code typically accesses many memory regions. Herein lies the difficulty: the programmer must determine what memory is accessed, and where it was allocated in order to allocate it with a tag. We have developed *Crowbar*, a pair of tools that gives programmers the information they need in order to partition existing code

using sthreads, rendering sthreads practical for use in partitioning legacy software.

4.1 Information needed by programmers

Before adding sthread support to an application, the programmer will typically have a mental model of the desired partitioning. For example, the client-handling code will most likely execute in an sthread, and authentication code will probably execute in a callgate. Identifying where in the code an sthread should start is a simple matter. For example, any code that follows the `accept` system call will most likely involve handling the client, and should therefore run in an sthread in order to contain attacks.

Once an sthread has been identified, the problems usually begin. The programmer needs to know all the memory regions and file descriptors used by the sthread, so that the privileges for these can be explicitly enumerated in the sthread's security policy at the time of the sthread's creation. There are usually only a few file descriptors, such as the client socket and a logging descriptor, and the programmer can typically identify these by studying the code manually. One can use SELinux's permissive mode to determine the SELinux policy, to give access to appropriate files, system calls, and allow signals. For memory permissions, though, the programmer would have to determine all the memory accessed by the sthread, and there are typically a very large number of such objects. The Apache client handler, for example, uses over 600 memory objects. Identifying these manually is not trivial. To continue the partitioning effort, once the programmer has identified an sthread, he needs to know:

- What memory is used by the sthread?
- For each memory object, what permission is needed (read/write)?
- For each memory object, where was it allocated? This way, it can be allocated in tagged memory, and access can be granted to the sthread.

Apart from identifying memory accesses needed by sthreads, the programmer may have some difficulty in identifying callgates. Callgates are commonly used to protect sensitive data, so the natural way of identifying them would be to determine all the code that uses a particular memory object. The programmer could manually determine the buffer which holds (say) a password. It is more difficult, however, for the programmer to manually determine all the uses of that buffer. Hence, once sensitive data has been identified, the programmer would like to know:

- For a particular memory object, which code uses it?

To aid in identifying sensitive data, the programmer may also want to know the following:

- In which memory regions does a particular function write?

Consider a function which "loads" sensitive data, such as a function that reads a private key from disk. In such cases, the programmer could easily identify the function loading the data, although he may have more trouble determining where the function writes the sensitive data. So to identify sensitive data, the programmer either identifies the buffer directly, or a function that is known to generate sensitive data. Once either has been determined, he can use this information to find the uses of this data, and hence, pinpoint the code to be executed in callgates.

4.2 Problematic design and complexity of legacy applications

Why does a programmer face difficulty when trying to determine what memory regions an sthread uses, and which code uses sensitive data (potential callgates)? Software today is almost always written in a monolithic fashion with no concern about protecting memory regions. Hence, all memory is available to all code. Programmers know this and rely on it.

To complicate matters, functions often use global variables and follow pointers between complex structures, touching many memory areas. It is seldom the case that a function only uses its arguments.[1] Therefore, when executing a function within an sthread, one cannot assume that only the arguments will be referenced and grant the sthread privilege only for the function's arguments. The invoked top-level function may in turn invoke many other functions, each of which may reference globals and other complex structures.

This problem therefore arises when partitioning monolithic applications that assume default-grant semantics, *i.e.,* all memory is available to all code, into compartmentalized applications that obey default-deny semantics, *i.e.,* no memory or file descriptors are available to code by default. One must enumerate all the permissions compartments require, and because the existing code was developed with "all available" in mind, there indeed are many. Doing this enumeration manually is tedious, as one must fully understand and read the code being compartmentalized (including inside all library calls). Fortunately, we can provide tools to ease this task.

[1] One exception is libraries, and indeed, applying sthreads at a library boundary proved simple, as we show in our Firefox and libPNG example in Chapter 5.

4.3 Approaches for determining partitioning information

Imagine how one can determine the permissions of an sthread in practice, manually. One can make a guess at the correct privileges for an sthread, although quite likely, the application will crash because of missing memory privileges. At that point, the programmer must use a debugger to determine which memory access caused the crash. Once it has been identified, the programmer must determine where the memory has been allocated, in order to covert a `malloc` call to `smalloc`. It could well be the case that this allocation occurred far away in the code from the memory access that caused the crash, forcing the programmer to hunt through many lines of code. Once privileges for that memory region have been granted to the sthread that needs them, the programmer can rerun the application, and repeat the above procedure for the next crash, and the next. For a substantial application, this cycle may repeat for many days, as was our experience, presented in Chapter 5, when partitioning Apache / OpenSSL manually.

Apart from impractical manual code study, two main approaches seem the most likely candidates for aiding in determining partitioning information: static analysis and dynamic analysis (run-time instrumentation). Static analysis can follow all possible code paths and determine all possible memory accesses. In practice, however, there are limitations on static analysis for C, such as the accuracy of "points to" analysis, *i.e.,* whether all function pointers and data pointers can be resolved. The complexities of static analysis for general C code were the factor that led us not to pursue static analysis. We opted for a simpler yet effective solution, albeit possibly a less powerful one.

Dynamic analysis involves analyzing a program at run-time. We can, for example, execute a program with some inputs and determine what memory regions are being accessed. Because we control the inputs when doing our analysis, we can "train" the program being analyzed with innocuous workloads only, in order to obtain the privileges needed for those clients only. This assures that we do not obtain (and hence grant) any extra privileges that may instead be required by an exploit. Hence, run-time instrumentation adheres closely to our default-deny philosophy: we start with no privileges, and consider only those inputs that we know are innocuous. Because we analyze only a fraction of inputs, we may not have captured all the privileges necessary for all possible production runs, so we suffer from a coverage problem. Although this is a limitation, we believe this to be a safer fail mechanism than allowing potential exploits through. Note that it is possible to detect the lack of privileges because of crashes in production runs, and we can therefore

incrementally fix applications, making them more robust with time. A combined approach of run-time instrumentation and static analysis is another option for future exploration.

4.4 Runtime inspection of data dependencies

To perform this run-time instrumentation, we developed *Crowbar*, a pair of tools to help the programmer partition existing code using sthreads. Crowbar supports the following three queries:

1. Given a function, which memory regions does it use, with which access modes (read/write), and where in the code was the memory allocated?

2. Given a list of sensitive memory objects, which functions use them?

3. Given a function, what memory does it write to?

The above information is most of what a programmer needs to know in order to partition existing code with sthreads. Missing information regarding system calls must be obtained via SELinux's audit log when running in permissive mode, and file descriptor privileges must be found manually. To obtain memory privileges with Crowbar, the programmer first manually identifies where to place an sthread, and defines a function that will contain the sthread's code. Next, he uses Crowbar's first query to determine the privileges the sthread needs, and which memory it uses that was allocated in other sthreads which must therefore be tagged for sharing. The programmer then must create callgates, and to do so, must identify memory that holds sensitive data. He either identifies a buffer that holds sensitive data manually, or locates a function that generates sensitive data. In the latter case, he uses the third query in order to determine the actual memory locations the function writes. Once these are found, he can use the second query to find all the uses of this memory—*i.e.,* the code to be run in callgates. The programmer no longer need hunt through and understand all the code. He only need identify key elements, such as where to place sthreads and buffers that hold sensitive data. The tedious tasks of determining all memory accesses, allocations and uses are handled by Crowbar.

We now describe the design of the tools and then give an example of their use. Crowbar is a pair of run-time instrumentation tools, developed using Pin [42]. Pin is a run-time binary instrumentation tool that allows modifying (instrumenting) code of an application as it is being run. The first of Crowbar's tools produces a run-time trace of the application. The second analyzes this trace. During the trace, Crowbar instruments and logs to disk every memory load and store. Each log

entry contains a complete backtrace of where the access occurred, so the programmer knows the calling context. This context is especially important for library calls, *e.g.,* `memcpy`, which may be invoked many times at different call sites. Reporting only the library function name (top stack frame) is uninformative since the programmer is interested in knowing the *caller* in order to determine which sthread is actually performing a memory access in the library function. Crowbar also keeps track of all globals, stack frames, and memory allocations (`malloc`). Thus, the resulting log relates every memory access to a heap allocation, global or stack frame. This information is particularly useful for heap accesses, since the programmer will not only know which buffer is being accessed, but also where it was allocated, so that the programmer can convert the appropriate `malloc` to `smalloc`. Once a log has been obtained, it can be queried multiple times to determine the desired information. The query tool is rather trivial, mainly acting as a pattern matching frontend like `grep`. A more advanced front-end could help the programmer further, by, for example, automatically opening source files at given locations.

4.5 Debugging secured applications

Crowbar includes additional functionality that is useful for applications that already use sthreads. When refactoring and changing sthread code, it is possible that the code stops working due to insufficient sthread privileges. Crowbar allows running an application in "permissive" mode to determine such errors. In permissive mode, all cross-sthread memory accesses are allowed regardless of the security policy, and any violations are logged. This mimics SELinux's ability to turn enforcement on or off. The advantage of implementing this functionality in Crowbar rather than (say) in the kernel is that Crowbar has more context, and can provide more information to the programmer. For example, Crowbar can determine where a buffer that caused a violation has been allocated—that would be impossible from the kernel alone, since memory allocations occur in userspace.

4.6 Implementation

We implemented Crowbar using Pin [42], a binary instrumentation tool. Pin allows dynamically modifying x86 code of an application, at run-time, as an application is executed. Thus we are able to instrument an application and log memory accesses to determine memory privileges required by code. Our Crowbar code consists of 3,800 lines of code. The tool keeps track of memory regions defined by a base and limit. Each memory region can either be a `malloc`ed buffer, a global or a stack

frame. To obtain a base and limit for a global, we require the program be compiled with debugging symbols, which provide the variable name, size and location. We instrument each function prologue and epilogue to keep track of stack frames. This also allows us to keep track of the current backtrace so that we can report it in log entries. We keep track of stack frames using the frame pointer, so we do not support binaries compiled with frame pointer omission. We also know when `malloc` and `free` are called by instrumenting their entry points, and can therefore register newly allocated memory areas. We then instrument every memory access, and determine in which memory region it lies, which in turn reveals which object is being accessed.

Mapping a memory access to a memory region is the most frequent operation that Crowbar performs, so we optimize this operation in several ways. First, we implement Crowbar as two processes: the first uses Pin to gather events, and the second analyzes them, for example, to determine which memory region is being accessed given an address. This allows us to exploit multi-core CPUs and achieve a higher throughput, as one core is busy executing the program and doing Pin's translation of basic blocks, while the other one is busy processing the results. One future further optimization would be to use SuperPin [76], which parallelizes the instrumentation. Second, we use *shadow memory* to speed finding a memory region given an address. We maintain a data structure that stores a pointer to metadata describing the memory region being accessed for each byte of the application's memory. This data structure allows us to look up a pointer's region in constant time.[2] To save space, one complication is how to store one 32-bit pointer per byte. Note that `malloc` always allocates memory in multiples of eight bytes, regardless of the requested size. Thus, in reality, we only need one pointer per eight bytes of memory and we can easily afford that much memory. Because globals can be a single byte in size however, our trick for `malloc` does not work, so we instead use a tree to store globals' allocation. This scheme performs well overall, since we found that the vast majority of accesses are on the heap and stack, rather than to globals.

Other optimizations involve reducing the number of instrumented memory operations. For example, when an operation is clearly a stack operation in the current frame, we need not instrument it; since an sthread is always allowed to access its stack there are no privilege violations to report. We detect these cases by checking whether a memory access is relative to the frame pointer. Another optimization is not instrumenting loader code, since all the memory the loader accesses (relocations) will always available to it, because of how we copy the memory map at `smain`, during sthread initialization.

To implement permissive mode, we emulate sthreads using standard pthreads.

[2] An alternative would be to keep regions in a tree but we found this approach slower.

This approach gives sthreads access to all memory, so that the application will never terminate due to insufficient privilege. We do, however, keep track of the privileges of the currently executing thread and log any violations that occur. We do not yet support copy-on-write memory in permissive mode, though. One possible implementation would be to check whether the accessed region is marked copy-on-write, and if so, copy it and redirect all future reads and writes to the copy. While the conceptual implementation of this is rather simple it does involve rather tedious bookkeeping and carefully written code (*i.e.,* a userspace page table and VM implementation). Despite this lack of copy-on-write support in our current implementation, we have still successfully been able to run applications that use copy-on-write, however. In practice, this is so because often only one run, *e.g.,* handle only one client's connection, is needed in order to determine the memory privileges that were violated. Since it is the first run, any memory marked as copy-on-write will hold its (correct) original contents, which will be replaced during the run. The *second* run could crash because that memory's content have unexpectedly changed, although we do not need a second run for applications in which each run yields the same set of privilege violations.

4.7 Limitations

As a run-time tool, Crowbar suffers from *coverage* limitations. Crowbar will only detect the memory accesses that occur during a particular run of the application it instruments. These are a subset of all possible memory accesses that could ever occur. As a result, insufficient privilege may be identified for sthreads, and potential crashes from protection validations may result when running the application in a production environment. To mitigate this problem, the developer must ensure sufficient coverage when producing traces. (Note that multiple traces can be combined, by simply concatenating trace files, in order to obtain a broader set of examples of memory access behavior.) This strategy is the same one often used when testing software, so an existing test suite for an application may be useful for producing Crowbar traces. Tools exist for measuring test coverage, so one can measure coverage objectively [2]. In our experience with OpenSSH and the SSL Apache servers, we found that only a couple of runs were necessary to produce traces that allowed running (without crashing) all our benchmarks and tests on the resulting partitioned applications. With OpenSSH for example, we required one trace in order to successfully login with each authentication mechanism supported. Note that exceptional cases, such as timeouts, authentication failures, and use of successive authentication mechanisms do work properly in our sthread-partitioned version, even though we did

not explicitly run test cases for these when using Crowbar. Looking at sthreads as a capability system, Crowbar tells the programmer which capabilities to give child sthreads. If an sthread is capable of authenticating the user using one mechanism or another, exceptional cases such as authentication failures are likely to work, and indeed they do in our OpenSSH version, since they require no additional privileges. These cases are handled *within* the sthread and an sthread is free to use its own private memory at will—it does not need any extra memory set up by the parent. Insufficient privilege problems typically occur at the transition from the master to the child sthread, since that is when tagged memory must be specified in an sthread's policy. We found that these problems are spotted early in the sthread's lifetime, for example, when it tries to access its arguments for initialization.

This termination problem—"when have we found all privileges?"—is not unique to Crowbar. If one were to produce an SELinux policy by using SELinux's audit log with enforcement disabled, the same problem would occur. One would anyhow need to ensure high coverage in order to be sure to discover all legal system calls and file accesses. Hence, Crowbar's run-time mechanism for determining permissions goes hand-in-hand with SELinux's.

Even though this is a limitation of Crowbar, it closely adheres to our principles of default deny. Crowbar's output is more restrictive, rather than more permissive. The programmer explicitly controls the inputs that an application should accept, and Crowbar permits only exactly those permissions. This approach allows a programmer to "train" Crowbar with "good" inputs, and it is likely that malicious inputs, such as exploits, will cause protection violations, due to sthreads' restricted privileges.

We intend to explore static analysis as an alternative to runtime analysis. Static analysis will yield a superset of the required permissions for an sthread, as some code paths may never be executed in practice. Static analysis would report the exhaustive set of permissions needed for an sthread not to encounter a protection violation. Yet these permissions could well include privileges for sensitive data that could allow an exploit to disclose that data. By using run-time analysis of the application running on an innocuous workload, the programmer learns which privileges are used when an exploit does *not* occur, but only those required for correct execution for *that workload*.

Crowbar is an aid to the programmer; not a tool that automatically determines a partitioning by taking security decisions on its own. Similarly, one must use Crowbar with caution. That is, one must assess which permissions should be granted to an sthread, and which need to be wrapped around a callgate. The tool alone guarantees no security properties; it merely responds to programmer queries as a programming

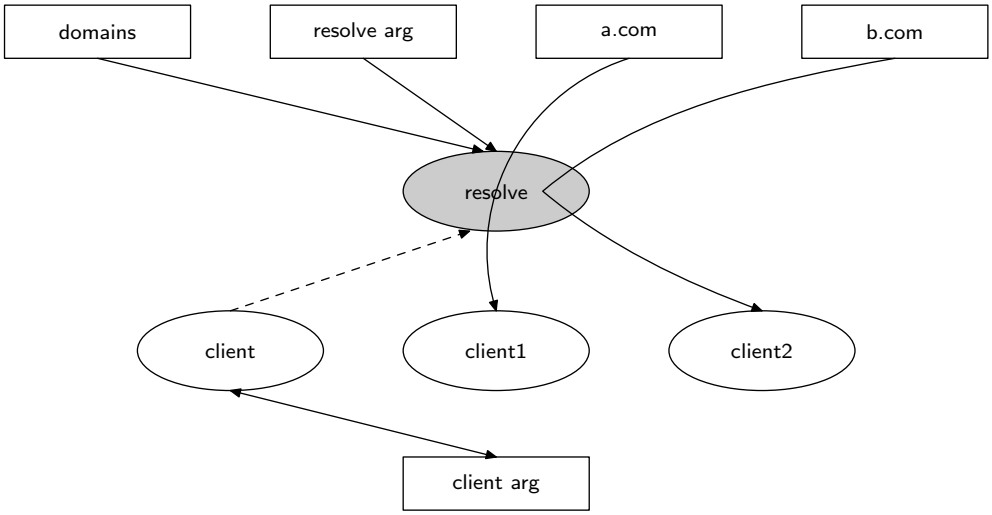


Figure 4.1: Static analysis of our DNS server.

aid, and it is the programmer who enforces correct isolation.

4.8 Visualizing the resulting implementation

We built an additional tool that can be useful to programmers once partitioning is complete. This tool analyzes source code and produces a diagram indicating the privileges of the code's sthreads and callgates, much like the diagrams we use when designing a partitioning. The intent of this tool is to visualize the resulting implementation in order to sanity check whether the programmer created and configured sthreads as expected in the secured application's design. This tool is not as useful as Crowbar, which helps a programmer realize a design, though it acts as a companion tool working in the reverse direction, since it can be used as a confirmation of whether the resulting code matches the design target.

Figure 4.1 shows the partitioning diagram produced from our static analysis for a DNS server that we shall detail in the next chapter. The main sthread is the `client` one, which has access to its arguments (passed by the master) and can invoke the `resolve` callgate. The `resolve` callgate has access to the arguments passed by the `client` sthread. It also has access to the table of zone files. Depending on the calling client, it can either access one zone or the other. The two sthreads labeled as 1 and 2 show these two different classes of clients and the different permissions in each case, *i.e.,* that their callgate can access only one of the two zones. One need not read the code to determine the partitioning that has been implemented.

The security of our system not only relies on callgates not being exploited, but also on sthreads being created properly, and on the whole partitioning being imple-

mented as expected. This tool could be used to check the latter property. We expect this tool to be of use in the deployment of sthreads in scenarios where there is one security architect responsible for determining the partitioning and another team for implementing it. The architect can then check the final implementation with the use of this tool, rather than by reading code.

We have implemented this tool using llvm [37]. We first reduce the program to a small working set, keeping only those functions that make use of the sthread APIs. We can then run a computationally expensive inter-procedural data-flow analysis on this much smaller program slice. We keep track of all possible values the security context for each sthread might take on. As shown in the DNS example, we handle branches and multiple possibilities for different security contexts for the same sthread correctly. Our analysis is rather simple, so it does not handle, though it detects, the case when a security context is set up via function pointers or by reassigning the security context pointer in arbitrarily complex ways. In fact, in all the code we wrote using sthreads, the security context setup is always a line of code just before sthread creation, and we never found a need to do this otherwise, so we expect this style to be common. Static analysis techniques can handle this style easily, and our implementation is complete enough to correctly compute the security context even if it is configured via multiple (direct) function calls, and in loops.

4.9 Summary

Partitioning existing code with sthreads is difficult because applications today are written in a monolithic style where all code can access all memory. To partition an application with sthreads, one must "carve out" sections of code that run in compartments, and determine all the many memory privileges that each compartment requires. This is very tedious to do manually, so we provide Crowbar, a tool that provides this information to the programmer. A sister problem is determining the users of sensitive data in order to factor out code that should run in privileged compartments. Crowbar helps in this aspect too, making the introduction of sthreads into existing code practical.

Chapter 5

Applications

We now assess whether sthreads and Crowbar are valuable tools by applying them to a range of applications and determining whether we can achieve demanding security goals. We applied sthreads to the following five applications: an SSL web server written from scratch, Apache & OpenSSL, OpenSSH, Firefox & libPNG, and a DNS server written from scratch. Our range of examples includes partitioning of both existing code and writing entirely new code with sthreads in mind. We can therefore compare our experience writing new code using sthreads with introducing sthreads into existing code with Crowbar's help. We present two implementations of the same service, one newly written, and another derived from existing code, so that we can comment on differences in the resulting security properties when writing new code versus when partitioning exiting implementations. We partitioned OpenSSH, the version prior to privilege separation, in order to check whether our system could be used to produce the partitioning that the original OpenSSH developers desired. We enhanced the security of Firefox & libPNG to gain experience with client-side applications and C++. Finally, our DNS example, a lightweight service, serves as a good performance benchmark, since sthread-related costs dominate. In the sections that follow, for each application we will describe our security goals, threat model, application design, and the security properties we achieved.

5.1 SSL web server written from scratch

We chose an HTTPS web server as a candidate for evaluating the use of sthreads in a complex application and in a threat model in which the attacker is powerful. Indeed, we show that we can protect sensitive user data not only if an attacker exploits sthreads, but also when the attacker acts as a man-in-the-middle and can both eavesdrop on all traffic and inject arbitrary packets between a legitimate client

and the server. To our knowledge, we are the first to consider such complex threat models, and we show that sthreads are flexible enough to defend against them. Interestingly, it is only once you are close to a least-privilege partitioning that such issues start to become relevant.

Our goal in securing the web server is to protect against the disclosure of sensitive user data, tampering with it, and injecting spoofed data. The main purpose of HTTPS is to protect the confidentiality of data; it is often used for credit card and other sensitive transactions. This motivates our choice of focusing on HTTPS rather than on HTTP only. In order to reach our end-to-end goal of protecting user data, there are a few other important subgoals that must be met. For example, the server's private SSL key must not be disclosed, or else an eavesdropper may decrypt traffic obtained from the network. Similarly, user session keys must be kept private. We now describe our threat model.

5.1.1 Threat model

We assume that the attacker can:

- Exploit any sthread instantiated for dealing with the attacker's connection and run arbitrary code in that context. sthreads were designed with this assumption in mind; they are unprivileged compartments. The attacker can therefore read or write any memory for which the sthread has the appropriate privileges. The same applies to file descriptors. The attacker can also invoke any callgate permitted in the sthread's policy, and call any system call allowed by the sthread's SELinux policy.

- Act as a man-in-the-middle. The attacker can modify or drop any packets sent between the client and server. The insertion of any packet in either direction is possible, too.

We assume that the attacker cannot:

- Exploit any callgate. We rely on this assumption since callgates are privileged compartments. If a callgate is exploited, the attacker has access to any resources (*e.g.,* memory, system calls) that the callgate can access, which typically include sensitive data.

- Influence the master process. We refer to the master process as the process that coordinates the creation of sthreads. This process does not read any network input, and so cannot be remotely influenced. It does, however, process local input, such as configuration files, and could therefore be exploited by these.

We consider only remote attacks, since a local attacker who can affect state such as configuration files has most likely gained enough privilege to divulge sensitive user data in other ways, such as by reading the SSL private key directly from the disk.

Given this threat model, we will now present a partitioning of the web server that will satisfy our goal of protecting sensitive user data.

5.1.2 Design

We shall explain our full design incrementally in order to facilitate understanding. First, we consider the case of a less powerful, passive attacker who can only eavesdrop on the network without injecting data. Then, we consider our full threat model, in which the attacker is active and has full man-in-the-middle capabilities, and thus can inject data, too.

No man-in-the-middle

To gain access to sensitive user data, the attacker must try one of the following options:

1. Disclose the data from the server by exploiting it. The attacker could try to exploit the web server and leak data of other users from the server's memory or disk. We can prevent this by running each client in a different sthread and holding the client's data in the sthread's private memory regions. We restrict disk access to sensitive files via SELinux. Thus, an attacker exploiting a web server will only gain access to his own private data, since he will be trapped in his own sthread.

   An attacker that is not a man-in-the-middle cannot exploit another session (hence sthread) by injecting an exploit via the network that appears to come from the other session. This is so because in this example we limit the attacker to be passive, from the network's point of view.

2. Disclose the data from the network by decrypting it. The attacker could eavesdrop on the network and try to decrypt the data. To do this, he would need the SSL session key. This session key could be obtained by knowing the SSL private key and eavesdropping on the SSL handshake. We need to prevent the server from divulging its SSL private key, or disclosing session keys of users.

To prevent the attacks listed, we need to meet the following criteria:

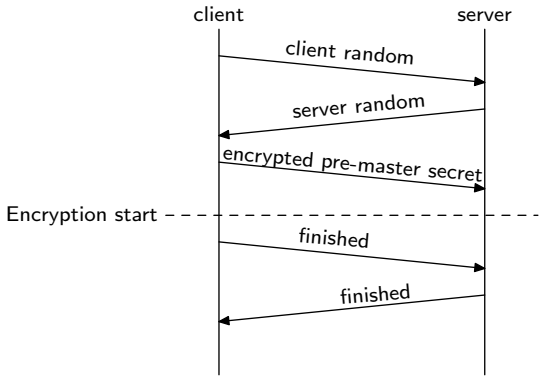


Figure 5.1: Simplified SSL handshake.

- Each client session must run in its own sthread, and sensitive user data must be kept private to the sthread.
- The SSL private key of the server must not be disclosed.
- The SSL session keys of other users must not be disclosed.

The last two properties are the more subtle ones and merit further attention. Consider how one may protect the SSL private key. For example, one may choose to hide it behind a "decrypt" callgate. An sthread invoking the callgate would not be able to determine the bits of the keys, although it will have access to an *oracle* and can *use* the key freely, which is tantamount to disclosing the key. One must therefore design callgates in such a way that their output is not sensitive or does not lead to such oracle attacks. A similar argument exists for a potential callgate that could be used to generate a session key—it must not generate the key of another session, else the attacker could generate the key for someone else's session and decrypt its traffic. To design these callgates in a secure fashion, we need to look closely at the SSL protocol to find a good match between callgate interfaces and the protocol in such a way that sensitive information is not disclosed by callgates.

The SSL handshake is the most important part of the protocol with regard to partitioning as it uses the SSL private key and sets up the session key. We considered the RSA handshake only. Its simplified operation is shown in Figure 5.1. To generate the session key, three inputs are needed: the client random, server random, and the pre-master secret. The pre-master secret is sent encrypted during the handshake and the server uses its private key to obtain the clear text. We cannot have a callgate simply decrypt the pre-master secret, or an attacker could eavesdrop on a handshake, exploit the server, supply the callgate with the eavesdropped encrypted pre-master secret, obtain its clear text, and calculate the session key based on the

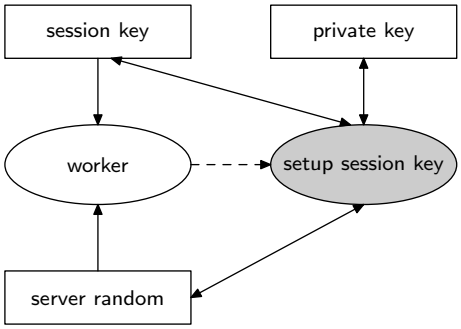


Figure 5.2: Protecting private key disclosure and arbitrary session key generation.

eavesdropped client and server randoms. Instead, we make the callgate return the computed session key directly. For this to be safe, the session key must not be that of another session, or an attacker could compute the session key of any session and decrypt traffic of all users. In other words, the session key must be unique for the attacker's session. The attacker must have no way of controlling its generation—*i.e.,* it must be a random key. Recall the three inputs needed to generate the session key: client random, server random, and pre-master secret. If the callgate takes only two of these inputs and generates the third, there is no way the attacker could generate the key for an arbitrary session. The server random fits nicely here as it is a random value, hence the attacker cannot control it, and it is generated by the server so it need not be supplied from elsewhere.

The partitioning shown in Figure 5.2 puts into practice the previous observations and protects against an attacker who cannot act as a man-in-the-middle. The `setup session key` callgate is the only one that has access to the private key. This callgate also generates the server random value, when first instantiated and invoked by the master (for each connection). The private key is fully protected as the output of this callgate is the session key, which cannot be reversed to the private key. The attacker cannot generate the session key of another eavesdropped session since he has no control over the server random. To get the session keys to match, he would have to brute force the prior connection's server random, which is unlikely given that it consists of 256 bits. Thus an attacker can only generate his own session key, which will allow him to decrypt his own traffic only. All user data is kept in the sthread's memory region and each session lives in a different sthread, hence the attacker cannot read other users' data from memory. The sthread abstraction therefore prevents one session from tampering with another one.

The partitioning described so far relies on the attacker not being able to inject packets containing exploits in an already established session. An active attacker could, for example, spoof a network packet and exploit the sthread of another session,

causing sensitive user data (or the session key) to be disclosed. We now show how we can protect against such active attackers, too.

Man-in-the-middle

We now consider the stronger threat model of an attacker who can exploit any sthread and fully control the network, including sending source-spoofed packets. When designing network servers with sthreads, it is customary to run every user session in a different sthread in order to avoid one user's learning (or modifying) another user's data. The assumption is that the "good" client will live in its own sthread, and the hacker will live in a different sthread, and thus have no way of accessing the memory of the good client. This assumption, however, does not hold when a man-in-the-middle is present. Consider a good client that establishes a connection to a server and is served by an sthread. A man-in-the-middle can inject an exploit appearing to originate from the good client, and so this exploit code will run in the good client's sthread. The attacker will have full control over the good client's sthread, and can arbitrarily leak (and modify) any sensitive data held in memory of the good client. Of course in a clear text network protocol a man-in-the-middle can read or write user data from the network without having to go through the trouble of exploiting the server. Cryptographic protocols such as SSL solve the man-in-the-middle problem by encrypting and authenticating network traffic. Will, however, SSL protect against man-in-the-middle attacks where the attacker can also (partially) exploit the server? Or, in other words, can we provide an implementation that is robust against such attackers?

To tackle this problem, a key observation is that an SSL connection consists of two parts: key establishment (the SSL handshake) and MACed communication. If we assume that the attacker does not hold the SSL session key, then the attacker can only act as a man-in-the-middle during the first part of the connection (the handshake), since that is the only time that unauthenticated clear text data is transmitted and will be accepted. Without the session key, the attacker cannot transmit valid MACed data during the second phase of the connection. Thus, in the second phase of the connection, we can assume an attacker who cannot inject exploits in an established connection, which yields security requirements that are easier to meet. Note that we assume that the attacker cannot exploit the MAC verification code.

Our high-level partitioning is shown in Figure 5.3. We create two sthreads, one for the unauthenticated clear-text part of the handshake and one for subsequent communication. The master's role is to start the SSL handshake sthread, kill it once

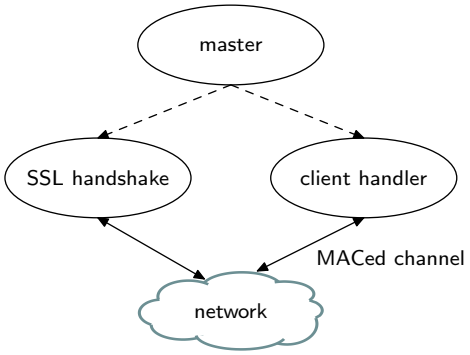


Figure 5.3: Two phase SSL protection.

authentication and encryption has been activated, and then start the client handler. The role of the SSL handshake sthread is to gather the inputs to the session key, returning these inputs to the master upon completion. Hence, the first phase is responsible for obtaining the client random, the pre-master secret, and ending when the change cipher spec message is received. Note that this sthread terminates before the actual SSL handshake is complete, but after authentication and encryption are turned on—the second-phase client handler sthread will complete the handshake. Once the first stage completes, the master creates a fresh client handler sthread to process the second phase (the authenticated and encrypted channel). In order for the second phase to deal with encrypted traffic, the master will supply it with the session key. This is generated by the master, which holds the private key, based on the inputs returned by the first phase sthread.

The second phase sthread deals with the user's (now encrypted) session and holds sensitive user data. Because it is a *new* sthread, the attacker must exploit it to gain control. To do so, he must inject an exploit via the network and pass the MAC check—*i.e.,* he needs the session key. The session key, though, is known only to the second phase sthread, and thus the attacker has no means of exploiting the second phase as he remains ignorant of the session key. The attacker could inject garbled cipher-text in an attempt to exploit the MAC verifier in the second phase. Clearly, we must trust this code, and so must audit it carefully, as if it executed in a callgate. We also shield the RSA key and session key generation behind callgates, as discussed earlier on protecting against non-man-in-the-middle attackers.

The attacker is free to exploit the SSL handshake sthread, since all communication is in clear-text the attacker requires no extra knowledge (*i.e.,* the session key is not needed). If the attacker exploits the SSL handshake sthread, though, the only way for the client to send (sensitive) data is to complete the SSL handshake successfully. The only way to do so is to signal the master to exit the first phase

and enter the second one. Thus, an attacker may either choose not to complete the handshake, and hence never obtain sensitive data from the client, or to complete the handshake, and hence lose control of execution because the SSL handshake sthread will be killed. Note that if the handshake is not completed, the attacker will not be able to calculate the session key as the pre-master secret (and private key) remain unknown to him. In both cases, the attacker will not be able to control the client handler sthread, where the sensitive data is.

5.1.3 SELinux policy

Table 5.1 shows the SELinux policies associated with the various components of the application: the master (managing sthread), the setup session key callgate, and the two client handling sthreads. The policy of the master sthread is most complex and privileged as it must setup and configure the network daemon. The master though does not directly process network input so it is unlikely to be exploited by a remote attacker. At a high-level, the master sthread needs sufficient privilege to load the executable and its libraries, and to setup a listening TCP socket on port 443 (HTTPS).

The setup session key callgate needs very little privilege as most of its computation is done without needing system calls. Only "/dev/urandom" is accessed to seed the random number generator.

The SSL handshake sthread handles user input so is most security critical. The policy is rather minimal, however, as those are the minimum permissions that any sthread would need to start running and access a client socket. First, the master is allowed to start (transition to) the SSL handshake sthread. The sthread is allowed to access some file descriptors from the master. Specifically, the sthread is allowed to write to standard output. (The webserver was started from within the program "screen", hence why screen's standard output file descriptor appears in the policy.) A production version would not allow the sthread to write to standard output, thus making the policy even more restrictive, but we keep this permission as we print useful debugging information. The sthread is also allowed to read and write to the TCP socket passed to it from the master (*i.e.,* the client).

We note that much of the SELinux policy can be simplified if sthreads are more closely integrated with SELinux. For example, sthreads already provide file descriptor protection so specifying permissions in an SELinux policy too, is redundant. Most importantly, however, it should be clear that the worker sthread has minimal access to the system: it can only access its own socket and standard output. This shows how splitting an application into multiple compartments not only reduces the

memory permissions required, but also greatly reduces the system privileges needed.

For the client handler sthread, we only show the differences in the SELinux policy from the base SSL handshake SELinux policy. In fact, almost all sthreads will have such a minimum policy like the SSL handshake one, so it acts as a common template and base case, which we shall omit from future policies we show. The most notable difference between the client handler and the SSL handshake sthread is that the former requires access to the filesystem to serve web content. Specifically, it can access all directories labeled `www_t` (ref. 27), and read any files with that label (ref. 28).

We note that the granularity of our partitioning was primarily dictated by memory permissions. If a compartment seemed to require too much memory access (*e.g.,* access to a private key) we split that compartment into multiple ones giving less access to each, likely shielding any private data behind a callgate. Partitioning, however, can also be influenced by SELinux policies. For tighter system call control, we expect further partitioning to be necessary, which may also decrease the application's performance. For example, if one of our goals was to serve dynamic content and prevent an attacker from obtaining the source code of (say) PHP scripts, we may require an additional callgate for processing the PHP script, thus denying the network handling sthread direct access to the actual PHP files holding the source.

5.1.4 Information revealed when exploited

We now summarize the information the attacker can obtain by exploiting sthreads in our partitioned web server. We demonstrate that this information is not sufficient to disclose private user data. Furthermore, we show how this information could have been gathered without exploiting the server. Hence, we do not give the attacker any extra information, or in other words, we do not give the attacker any extra benefit upon successful sthread exploitation—we fully contain attacks.

To determine the information that an attacker has access to, we look at the memory regions an sthread can read directly, and the output of callgates or other trusted code (*i.e.,* the master). The following information is available to the attacker:

server random. This is available to the SSL handshake sthread, and is not sensitive, as it can be eavesdropped from the network.

master secret. This is available to the client handler sthread, and is the only piece of information not obtainable via the network. Because this information is only available in the second phase, only attackers with valid session key information (*i.e.,* non-man-in-the-middle) can obtain it. The master secret consists of a hash derived from the pre-master secret. It does not yield any

Ref.	**Master**
1	`allow apm_master_t nfs_t:file { entrypoint read getattr execute };`
2	`allow apm_master_t sysadm_devpts_t:chr_file { read write getattr };`
3	`allow apm_master_t sysadm_screen_t:fd { use };`
4	`allow apm_master_t etc_t:dir { search };`
5	`allow apm_master_t lib_t:dir { search getattr };`
6	`allow apm_master_t ld_so_cache_t:file { read getattr };`
7	`allow apm_master_t usr_t:dir { search };`
8	`allow apm_master_t nfs_t:dir { search getattr };`
9	`allow apm_master_t shlib_t:file { read getattr execute };`
10	`allow apm_master_t lib_t:lnk_file { read };`
11	`allow apm_master_t sysadm_t:fd { use };`
12	`allow apm_master_t ld_so_t:file { read };`
13	`allow apm_master_t self:tcp_socket { create setopt bind listen accept };`
14	`allow apm_master_t http_port_t:tcp_socket { name_bind };`
15	`allow apm_master_t inaddr_any_node_t:tcp_socket { node_bind };`
16	`allow apm_master_t self:capability { net_bind_service };`
17	`allow apm_master_t netif_t:netif { tcp_recv tcp_send };`
18	`allow apm_master_t node_t:node { tcp_recv tcp_send };`
19	`allow apm_master_t port_t:tcp_socket { recv_msg send_msg };`
Ref.	**setup session key**
20	`allow apm_ssk_t urandom_device_t:chr_file { read getattr };`
Ref.	**SSL handshake**
21	`allow apm_master_t apm_phase1_t:process { dyntransition };`
22	`allow apm_phase1_t apm_master_t:fd { use };`
23	`allow apm_phase1_t sysadm_screen_t:fd { use };`
24	`allow apm_phase1_t sysadm_devpts_t:chr_file { write getattr };`
25	`allow apm_phase1_t apm_master_t:tcp_socket { read write };`
26	`allow apm_phase1_t apm_master_t:process { sigchld };`
Ref.	**client handler**
27	`allow apm_phase2_t www_t:dir { search };`
28	`allow apm_phase2_t www_t:file { read };`

Table 5.1: SELinux policy for our web server written from scratch.

Component	Line count	Percentage
Total	2,291	100%
sthreads	1,447	63%
Callgates	249	11%

Table 5.2: SSL web server line count.

information about the RSA key (because of hashing). Furthermore, it yields no new information to attackers, because in order to obtain it from the client handler sthread, an attacker would already have had to know the session key (and hence master secret).

MD5 and SHA1 state. This is available to the client handler sthread. The last two messages of the SSL handshake (finished) consist of the MD5 and SHA1 hashes of all the messages sent during the SSL handshake. Because the SSL handshake is split across two sthreads, the first sthread needs to export a partial hash computation to the next one. This information is not secret and can be computed by eavesdropping the network and hashing the messages exchanged during the SSL handshake.

5.1.5 Avenues for exploitation

The security of the proposed web server relies on there being no exploits for callgates. To evaluate whether we have truly narrowed the problem, we assess the chance that a callgate is exploited. First, we use a line count metric to compare the number of lines of untrusted code (sthreads) with the number of lines of trusted code (callgates). This comparison gives a rough approximation of the attack surface, if we assume that all single lines of code are equally likely to be exploited. Then, we discuss the harm an exploited callgate can do, and how a callgate can possibly be exploited by inspection of its inputs.

Table 5.2 shows line counts for the sthreads and callgates in our web server. We only count the lines of the application and not of libraries (*e.g.,* libc, OpenSSL). We therefore trust libraries used in callgates, and the kernel, and do not account for their size. Our metrics represent the attack surface only in the application itself. In reality an attacker can choose to exploit the kernel, or when targeting a callgate, a library used by it. A more robust implementation would apply sthreads within libraries, or at least attempt to wrap library calls in lower-privilege sthreads in order to isolate vulnerabilities in libraries when invoked from a callgate.

Our web server consists of 2,291 lines of code. Of these, only 11% of the code lies in callgates and hence this is the only code, with respect to the application, that

```
void calculate_master_key(uint8_t *pre_master, uint8_t *cli_rand);
void read_ssl(int sock);
void session_master_key(uint64_t session_id, uint8_t *cli_rand);
```

Figure 5.4: SSL web server callgate interface.

we must audit. Most of the code—63%—is unprivileged, and runs in sthreads. The remaining code (26%) is the master, which coordinates the sthreads. This code is not at risk since it does not process any network input and hence it is highly unlikely that it can be exploited by a remote attacker. All of the network inputs passed to the master (*e.g.,* client-random) are relayed opaquely, without any processing, to the relevant callgates. The same occurs with exported data from the SSL handshake to the client handler—the master does not *process* it so it very unlikely that the master gets exploited by this data.

Our second metric for assessing exploit probability is analyzing the interface to callgates. Figure 5.4 shows the callgate interface for our web server. There are three pieces of trusted code in our SSL web server implementation: one for master key generation, one for input MAC verification in the client handler, and one that implements session caching. The master key generator takes two inputs of fixed length: the encrypted pre-master secret, and the client random. It knows the server random already since the master generates it. We argue that there is little space for attack here, since the function expects two random values and handling linear buffers of fixed length is relatively simple. The MAC verifier reads from the network, decrypts, computes a hash and checks it. Thus we trust a decryption routine (*e.g.,* RC4) and a hash routine (*e.g.,* MD5). Both of these are expected to work with random inputs, and their input is a linear buffer (start pointer and length). The routines do not perform any complex parsing and have been designed to take random values as input so should be less likely to be exploited. The final routine deals with session caching, and merely looks up a session ID in a hash table, computing the master key for the given client random. This last callgate shares nearly all code, apart from the session lookup code, with the master key calculation callgate, hence the attack surface is increased very little.

The last avenue for attack is via the exported state from the SSL handshake to the client handler. This consists of MD5 and SHA1 state. We believe that this state can be sanity-checked since it consists of a fixed-size array and indices pointing to it. The state array is designed to work with random values anyway, so it should require checking valid indices only.

5.2 Apache & OpenSSL

We partitioned Apache (v1.3.19) & OpenSSL (v0.9.6) to compare the resulting code structure with that of our own SSL web server implementation. Being relatively large and complex codebases, Apache & OpenSSL also give us insight into applying sthreads and Crowbar to real legacy code. As with our SSL web server, we have the same goal of protecting sensitive user data and being resilient to the man-in-the-middle threat model. In the sections that follow we describe the design of our Apache & OpenSSL partitioning and its security properties, commenting on differences from our own SSL web server implementation.

5.2.1 Design

When writing code from scratch, we have much more freedom as to where to place sthread boundaries, as we discovered when comparing our web server implementation with our SSL Apache partitioning. The key difference between the two is when we switch from the first phase (the SSL handshake) to the second phase (MACed channel). Referring back to Figure 5.1 one can see that encryption is turned on before the SSL handshake completed. With Apache & OpenSSL, we split right after the SSL handshake completes—that is, after the finished messages are sent. The finished messages are encrypted hashes of all previous messages, used to ensure that nobody tampered with the handshake and that the correct keys are established. Splitting after the entire handshake is a natural boundary when dealing with OpenSSL, since its function `SSL_accept` deals with the handshake, and `SSL_read/write` with the rest of the connection. From a protocol perspective, though, we want to switch between the two phases once encryption is turned on, *i.e.,* just before the finished messages, though still during the handshake. This is where the split occurs in our from-scratch web server implementation. Putting the split just before encryption starts is advantageous, since the SSL handshake does not need to know anything about encryption or decryption, and hence needs no access whatsoever to the session key. This is consistent with our requirement that the session key must not be disclosed during the SSL handshake.

In Apache, the SSL handshake sthread is more complex, since it needs to perform the whole handshake, and hence needs some access to the session key in order to handle the finished messages. Figure 5.5 shows the compartmentalization of the handshake phase. The goal is to not disclose the session key to the SSL handshake sthread, hence only callgates have access to it. The session key setup is the same as the one used in the non-man-in-the-middle attacker threat model, although we do not return the session key to the sthread. The session key is used twice during the

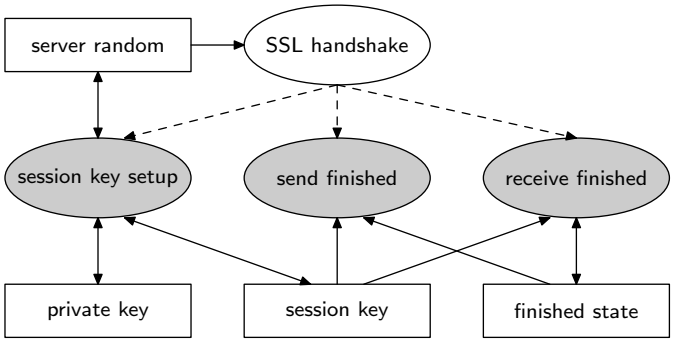


Figure 5.5: First phase partitioning of Apache & OpenSSL.

handshake, namely during processing of the finished messages, to validate whether a sane handshake has been completed and the two parties agree on the established key. The `received_finished` callgate returns no output to the sthread, so it cannot yield information regarding the session key. The `send_finished` callgate returns an encrypted hash of all the handshake messages, and since its output is a hash of the input, it cannot be used as an oracle to encrypt arbitrary data. Thus, an exploit in the first stage will reveal no information regarding the session key and the attacker has no oracle for it.

Note the difference in state exported from the SSL handshake sthread to the client handler sthread in our two implementations. With Apache, we export cipher state since the SSL handshake sthread will need to perform some encryption and decryption (finished messages). There is no export of hash state since the finished messages are dealt with entirely in the SSL handshake sthread. In our "from scratch" implementation instead, we export hash state but no cipher state. One advantage of the latter approach is that the hash state is always the same (MD5 and SHA1) regardless of the cipher suite selected. If cipher state had to be exported, it would have to depend on the the selected cipher, and would thus make state export more complex. This exported state is probably the most complex data structure passed between the two phases, and hence the most likely avenue of attack, so it makes a real practical difference if it is minimized or simplified. Our from-scratch implementation definitely wins here.

5.2.2 SELinux policy

Table 5.3 shows the SELinux policies for the worker sthreads in Apache. For clarity we omit standard sthread permissions required for sthread invocation (dyntransition), signaling completion (SIGCHLD), and reading / writing to standard input / output and the client's socket. We also omit the policy for the master sthread which

Ref.	**session key setup**
1	`allow app_sks_t urandom_device_t:chr_file { read };`
2	`allow app_sks_t www_cache_t:dir { search };`
3	`allow app_sks_t www_cache_t:file { lock read write getattr };`
Ref.	**SSL handshake**
4	`allow app_ssl_accept_t etc_t:dir { read search };`
5	`allow app_ssl_accept_t locale_t:file { read getattr };`
6	`allow app_ssl_accept_t www_log_t:file { append };`
Ref.	**client handler**
7	`allow app_request_handler_t www_t:dir { search getattr };`
8	`allow app_request_handler_t www_t:file { getattr read };`
9	`allow app_request_handler_t self:process { signal };`

Table 5.3: Apache SELinux policies.

is code we inherently trust (in fact, launched as root).

The session key setup callgate needs access to "/dev/urandom" to generate the server random and also needs to access the session key cache, stored on disk. The SSL handshake sthread requires logging capabilities (ref. 6) as it logs the ciphers negotiated in ssl_log. Locale information is needed (ref. 4 and 5), for example, to store log dates according to local conventions. The client handler sthread additionally requires reading files marked as `www_t` (ref. 7–8) to serve web content. It also modifies its signal handlers, for example, to set alarms and alter the behavior of SIGPIPE.

Once again we note that the SELinux policies remain strict. They are remarkably similar to the policies needed for our hand written web server. SELinux policies are much coarser grained than sthread memory permissions. Thus, we expect applications with the same functionality to have very similar SELinux policies. This does not hold with memory permissions as it depends on the complexity and internal structure of the application, which can be substantially different even though the end-to-end functionality of the application remains the same.

Our session key setup SELinux policy allows reading and writing to a session cache file. This could be one place where a callgate (or sthread) can be further split to further restrict an SELinux policy—one callgate can generate the session key, without access to the session cache, and another one can populate the session cache. However, in practice, this would not improve security much since an attacker able to exploit session key generation will obtain the server's private key, which will unlock any eavesdropped traffic. Any leaked (cached) session keys would not provide more information in this context, as they can be calculated by using the private key and eavesdropped sessions.

Component	Line count	Percentage
Apache+OpenSSL total	252,030	100%
sthreads	60,844	24%
Callgates	15,769	6%

Table 5.4: Apache & OpenSSL line counts.

```
void session_key_setup(SSL* s);
void receive_finished(SSL* s);
void send_finished();
void SSL_read(SSL* s);
```

Figure 5.6: Apache callgate interface.

5.2.3 Information revealed when exploited

We now evaluate what information is available to an attacker who exploits the sthreads. During the SSL handshake, the attacker can learn the server random and the encrypted finish message. This information could have been eavesdropped from the network so the attacker gains no extra benefit from exploiting the server.

During the second phase, the client handler has access to sensitive user data and the session key. Due to the protection of the MAC, the client handler can only be exploited if the session key is known, hence, only a legitimate client can exploit his own session. Such a client would learn his own traffic and session key, which is information already known. An external attacker cannot gain this information since he lacks the session key.

5.2.4 Avenues for exploitation

We now evaluate the attack surface of our implementation. Table 5.4 shows line counts for relevant parts of the code. In our implementation sthreads comprise about 60,000 lines of code and our callgates total approximately 16,000 lines. The rest of the code consists of the master, which is not under threat as it does not interact with the network, and OpenSSL cipher code not used in HTTPS. With this partitioning we therefore narrowed the problem to a quarter of the original size. In other words, we must manually audit about 16K lines of code, rather than ≈76K.

We have a total of four trusted components and their interfaces are shown in Figure 5.6. The `session_key_setup` callgate, if exploited, will reveal the server's private key, hence this is the most privileged callgate. It takes as input the client random and encrypted pre-master secret, both of which are fixed in length and

random values. The `receive_finished` callgate expects encrypted data, and we assume that the MAC verification routine is unexploitable, so the attacker must know the session key to inject an exploit into `receive_finished`. If the session key was known, exploiting the callgate would not yield much benefit since it has access only to the session key (known to the attacker), and not the server's private key. The `send_finished` callgate takes no input from the sthread, although it takes the output of `receive_finished` as input. Thus, an attacker would need to exploit `receive_finished` as well in order to also exploit `send_finished`—unlikely based on our argument for `receive_finished`. Finally, we trust that `SSL_read` drops packets with a bad MAC. This involves reading from the network, computing a MAC, and comparing its value to that in the packet. We argue that this code can be audited and its input is rather limited—an SSL packet, of which the maximum and received sizes are known. Also, MAC verification is one of the first operations that occurs, so we only trust the beginning of `SSL_read`'s code.

To validate that our Apache implementation indeed protects the private key against basic attacks, we dump the entire memory of all callgates and sthreads, by reading /proc/self/mem. We then search for the private key bits, and indeed we only find them in the session setup callgate callgate memory dump. This emulates an attack exploiting sthreads and search for the private key in memory, and this attack is indeed blocked by sthreads. Access to the private key via the filesystem is blocked too, thanks to SELinux, though standard UNIX file permissions would have sufficed.

5.2.5 Past exploits

The version of Apache/OpenSSL we partitioned suffered from a past exploit [1]. The vulnerability lay in OpenSSL's code, when the pre-master secret was read from the network. First, the length of the pre-master secret was read from the network, and then those many bytes were read from the network in a fixed buffer. Thus, an attacker could supply a large value for the key length, and the network parsing code would overwrite memory. In our partitioning, the network handling (and parsing) code lies in the SSL handshake sthread. Thus, such attacks would be fully contained.

At the time of the Apache/OpenSSL exploit, there was also a `ptrace` kernel exploit [69], allowing remote Apache attackers to gain root access (Apache typically runs as nobody). sthreads provide no guarantees against kernel exploits so it is important to remember that despite sthreads protecting applications, attackers could still fully compromise the machine via kernel vulnerabilities. For this particular exploit, SELinux would have blocked `ptrace`, preventing the attack. In general

Vulnerability	Location
Client key overflow [43]	SSL handshake sthread.
Session ID buffer overflow [71]	SSL handshake sthread.
OpenSSL ASN.1 Parsing Vulnerabilities [49]	SSL handshake sthread.
RSA private key timing attack [13]	Covert channel—not protected.
Insecure Protocol Negotiation Weakness [5]	Design error—not protected.
CBC Error Information Leakage [11]	Side channel—not protected.
Bad Version Oracle Side Channel Attack [73]	Side channel—not protected.

Table 5.5: Past OpenSSL vulnerabilities.

though, other exploits such as `brk` [67] would have worked as such system calls are allowed.

We note that an exploit anywhere in Apache's code will not allow reading the server's private key, since only OpenSSL's callgate has access to that information. Thus all past Apache exploits will not affect the privacy of the server's private key. We therefore examine OpenSSL exploits to determine which would compromise our security goals, such as protecting the private key. Table 5.5 shows OpenSSL's past vulnerabilities that have working non-DoS remote exploits applicable to Apache, and indicates in which compartment the vulnerability would occur, and whether sthreads help to stop exploits.

5.2.6 Discussion

The main problem with our SSL Apache implementation is that it passes around OpenSSL structures, which are indeed complex, from untrusted to trusted components. We used them in order to quickly reach a working implementation, through minimal changes to the existing code. In fact, we only changed 1,700 lines of code. This came at the cost of security, though. If we made more intrusive changes, we could have created callgates with more restrictive interfaces.

The size of the trusted code is much larger in this implementation compared to our from-scratch web server since in the latter case, we wrote it much more tersely to do exactly (and only) what we want. When working with Apache & OpenSSL we were aiming for our changes to be minimally intrusive (small changes) although that came at the expense of working on higher level APIs which include more code. Also, in the case of our newly written web server, the main benefit of writing callgates from scratch is that we can control their interface and define exactly what the input data structures look like. With Apache, we were tied to OpenSSL data structures which, being generic, can be more complex than necessary in this specific case. By having

a simpler interface to callgates, the likelihood of their being exploited is reduced.

The inputs to callgates and state export from the SSL handshake to the client handler is much cleaner in our from-scratch web server than in our Apache & OpenSSL partitioning. In the former case we could specify small and simple data structures that can be easily checked. In the latter case we were forced to use existing data structures with which we were unfamiliar. One thing to note, though, is that perhaps we could use our handwritten callgates in the Apache version by writing a small adaptation layer. This step would assure us further that the callgates are less likely to be exploited, because we could better sanity-check their inputs. This will also reduce the size of the callgates, because we can include only code that is strictly necessary for the callgate's operation.

5.3 OpenSSH

Modern OpenSSH implementations are privilege-separated using UNIX processes and pipes [57]. We partitioned an old version of OpenSSH (v3.1p1) that dates from prior to privilege separation, in order to determine whether sthreads would have been sufficient to meet the security goals the original developers had in mind when securing OpenSSH. Furthermore, we wanted to see if our design would differ.

5.3.1 Threat model

Our security goals are the same as those of the OpenSSH developers. Specifically:

- Exploiting the pre-authentication phase must be harmless. The attacker must not gain access to any data from other sessions, will be trapped in an empty `chroot`, and have an unprivileged user ID.
- The attacker must not be able to obtain the server's private key.
- The attacker must not obtain privileges without authenticating. The attacker must not skip authentication or obtain privileges different from those allowed by his userid's credentials. Valid credentials must be supplied and only the corresponding permissions will be yielded.
- The attacker must not observe other users' sessions.

Our threat model is the same as that of the OpenSSH developers. That is, untrusted compartments (sthreads) can be exploited and trusted ones (callgates) cannot. In OpenSSH's terminology, the *monitor* cannot be exploited whereas the *child* can. We do not protect against man-in-the-middle attacks in our OpenSSH

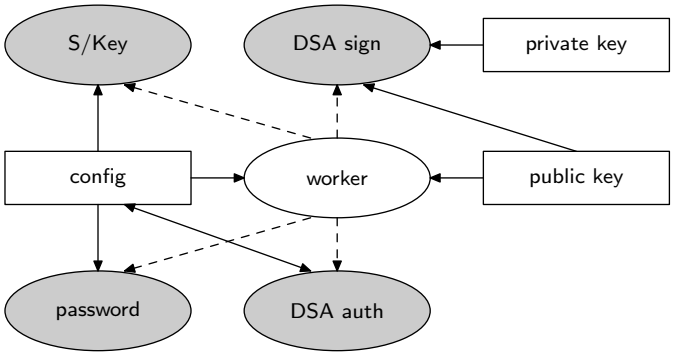


Figure 5.7: Partitioning of OpenSSH.

example, and indeed many of the attacks outlined for Apache would succeed with OpenSSH. One could of course go the distance with OpenSSH, too, in order to thwart even those complex attacks.

5.3.2 Design

Figure 5.7 shows our OpenSSH partitioning. We use an sthread to handle each connection. In order for the server to prove its identity it must sign data. We allow the sthread to sign data via the `DSA_sign` callgate. The sthread will not be able to obtain the server's private key (a goal) although it will have access to a signing oracle (not a goal). To authenticate, a different callgate is used depending on the authentication mechanism. We currently support DSA key authentication, S/Key and standard password authentication.

Once the sthread authenticates itself, its privileges are escalated by the callgate that was used for performing the authentication (*e.g.,* password). Initially, the sthread runs with a restrictive SELinux policy. Upon successful authentication, the callgate changes the calling sthread's SID (SElinux ID / policy) to the user's specific SID.

This is quite different from how privilege-separated OpenSSH works. Privilege-separated OpenSSH, to be portable across different UNIX platforms, creates another post-authentication child and exports state from the first child to the second one. This mimics what we did in our web server implementations. With our web servers, we were forced to do so because we assumed that SSL handshake could be exploited, and relied on the client handler's not being exploited (by an man-in-the-middle), and so needed a pristine sthread for the second phase. In OpenSSH's case, we do not require the post-authentication stage to remain unexploited, because the user (or attacker) must have provided valid credentials. If credentials were known, one can connect normally to the server and execute arbitrary code from the shell—there is

Ref.	**DSA sign**
1	`allow ssh_sign_t urandom_device_t:chr_file { read getattr };`
Ref.	**S/Key**
2	`allow ssh_skey_t etc_t:dir { search };`
3	`allow ssh_skey_t skey_t:file { read write };`
Ref.	**DSA auth**
4	`allow ssh_dsa_t home_t:dir { search getattr };`
5	`allow ssh_dsa_t home_t:file { getattr read };`
Ref.	**Password**
6	`allow ssh_pass_t etc_t:dir { search };`
7	`allow ssh_pass_t etc_t:file { read getattr };`
8	`allow ssh_pass_t shadow_t:file { read getattr };`
Ref.	**worker (pre-authentication)**
9	`allow ssh_child_t urandom_device_t:chr_file { read getattr };`
10	`allow ssh_child_t var_t:dir { search };`

Table 5.6: OpenSSH SELinux policy.

no need to exploit the server. Hence, we can use a single sthread and change its privileges once it proves possession of valid credentials.

There is no way to skip authentication but still obtain privileges, because only an authentication callgate can assign privileges, and it must be invoked with valid credentials in order for it to grant privileges. Also, it is impossible to provide valid credentials for one user and obtain the permissions of another one, since the callgate checks credentials and gives permissions in one atomic operation from the sthread's point of view. The only way to evade authentication would be to exploit a callgate, which is disallowed by our assumptions.

We give the worker access to configuration data, since it needs to know which authentication mechanisms are allowed, which protocol version to use and so on. The authentication callgates access the configuration data to validate whether the called authentication mechanism is allowed by the configuration, and to obtain any other parameters needed for authentication (*e.g.*, filenames used for keys).

5.3.3 SELinux policy

Table 5.6 shows the SELinux policies for OpenSSH. For brevity, the policy of the master is omitted, together with any rules required to allow transitioning into an sthread. The worker sthread is also allowed to read the client's TCP socket, as per the minimum sthread policy needed for network daemons. The worker's post-authentication policy is SELinux's generic user role policy, which, for example, allows the user to access his own files.

The callgates have all very similar policies, where the only difference is which files they can access. They all need to access a password or key database, each in different locations, ranging from /etc/passwd, to a user's public key in the user's home directory. Pre-authentication, the worker needs not any special privilege as authentication is performed by the callgates, and shell execution occurs post-authentication, when the callgates cause the worker to switch to the user's own SELinux policy.

5.3.4 Information revealed when exploited

We now discuss the information available to an attacker who successfully exploits the worker sthread. First, the attacker has knowledge of the configuration file. This could be considered sensitive, although that can be fixed by giving the worker a limited view of the configuration. The limited view could contain only those configuration aspects that may be determined from the network by probing the server. For example, a remote attacker could try different authentication mechanisms and see which ones are allowed by the server; this would effectively leak bits of the configuration. In any case, protecting the configuration was not one of our original goals, as we did not deem it being of paramount importance.

Second, the attacker has access to a signing oracle. We have not considered man-in-the-middle attacks in our OpenSSH implementation, so we ignored any potential vulnerabilities due to oracles. If these were a concern, techniques similar to those used in our web server implementations would address it.

Finally, there is a subtle third point that is worth mentioning. Some authentication mechanisms require *two* callgate invocations—for example, one for the username and one for the password. This may be necessary in order to compute the S/Key prompt, or check whether a user is allowed to log in. If two calls are used for authentication, it is important that the first call does not leak any information. For example, the callgate implementation might decide to return *null*, if a user does not exist. This would allow an attacker to enumerate usernames via an exploited sthread, which would not have been possible from the network alone, since a password prompt is always returned. Thus, the attacker can learn more from exploiting the server than from the network alone, which is an indication of a poor partitioning. Such vulnerabilities have occurred in practice with OpenSSH. For example, the S/Key prompt would not be returned to the network if a user did not exist [58]. This problem has been fixed by returning a fake prompt no matter what. However, in the latest version of OpenSSH (v4.7) at the time of this writing, when requesting a username, the monitor process still returns *null* to the child. This is a potential vulnerability since an attacker that exploits the child could still enumerate user-

Component	Line count	Percentage
Sthreads and callgates	26,114	100%
Sthreads	24,564	94%
Callgates total (privileged)	1,550	6%

Table 5.7: OpenSSH line count.

```
void dss_sign(uint8_t *data, int len);
void auth_getpwnam(char *user);
void auth_password(char *pass);
void auth_key(chr *user, Key *key);
void auth_skey_user(char *user);
void auth_skey_pass(char *response);
```

Figure 5.8: OpenSSH callgate interface.

names. We believe that it is important to push the line of defense all the way to the trusted components (callgates and monitor) and not only to the network, as OpenSSH does. Hence, in our implementation, not only does the network leak no information on whether or not a user exists, but also the callgates do not divulge such information, because they always return valid (or fake) information in response to username requests.

5.3.5 Avenues for exploitation

We now examine the likelihood of exploiting our OpenSSH implementation. Table 5.7 shows line counts for our implementation. Our code consists of ≈1,500 lines in callgates and ≈24,500 lines in sthreads. We have reduced the attack surface by 94%.

To further assess the threat posed by callgates, we must study their inputs, shown in Figure 5.8. The password and S/Key callgates both take a username and a password (or response). We believe that these two inputs can easily be sanity-checked to avoid exploits, since they consist of two text strings. The DSA sign callgate takes a buffer of random data and computes over it. Given that this function is expected to work with random data we believe that it ought to be resilient against exploits, since an exploit can be classed as an instance of a random input. The DSA authentication callgate is the one of greatest concern, since it takes a key as a parameter and performs parsing on it. We did narrow the attack surface to ≈1,500 lines, so it should be possible to manually audit the code, paying special attention to the DSA callgate.

Vulnerability	Location
CRC-32 exploit [80]	worker sthread.
Challenge/response exploit [27]	worker sthread.
Channel Code Off-By-One Vulnerability [54]	worker sthread.
Existing password weakness [28]	Covert channel—not stopped.
Root authentication timing [7]	Covert channel—not stopped.
Authentication Execution Path Timing Information Leakage [31]	Covert channel—not stopped.

Table 5.8: Past OpenSSH vulnerabilities.

5.3.6 Past exploits

The version of OpenSSH we partitioned suffered from a past exploit [27]. The problem was an integer overflow when reading the responses from a challenge / response authentication mechanism, such as S/Key. The integer overflow caused a smaller-than-necessary memory area to be `malloc`ed, causing a heap overflow. This occurs when receiving data from the network, which in our partitioning is handled by the worker sthread. Hence, the exploit would occur there, so our partitioning would mitigate this vulnerability.

As a proof-of-concept we ran OpenSSH's integer overflow exploit on vanilla OpenSSH and on our sthread version. On vanilla OpenSSH, the exploit yielded a root shell—the shellcode ran and had sufficient privilege to execute "/bin/sh". On our sthread version, the exploit failed. The shellcode was able to run though it was unable to execute "/bin/sh" because of the SELinux pre-authentication policy denying this. We modified the exploit's shellcode to read all memory available to the process and return it on the network, in an attempt to disclose the server's private key or any cached contents of password files. As expected none of this information was found because the sthread does not have access to such memory. This other exploit variant succeeded on vanilla OpenSSH (prior to privilege separation) where not only the server's private key was leaked, but also hashed user passwords as stored in the shadow file were disclosed. Privilege separated OpenSSH fails the exploit.

Table 5.8 shows OpenSSH's past vulnerabilities that have working non-DoS remote exploits on default OpenSSH configurations, indicating where they would lie in our partitioning. We omit and cannot comment specifically on PAM-based exploits as we have not yet added PAM support to our sthread OpenSSH.

5.3.7 Comparison with privilege-separated OpenSSH

Sthreads are flexible enough to accommodate the partitioning required for privilege-separated OpenSSH. We now discuss differences between our design and approach and today's privilege-separated OpenSSH. Our design avoids creating two children and exporting information between them, resulting in a much simpler implementation. This is possible thanks to SELinux, with the caveat that the resulting implementation is non-portable. Our system does not limit us to such a design, and in fact, we could split OpenSSH into two sthreads, just as we did with Apache, if it were necessary for whatever reason. Another difference in our approach is that we put the emphasis on not leaking information from callgates, rather than from the network, pushing the line of defense higher. For example, we return syntactically valid user information from our authentication callgates, even if a user does not exist, rather than *null* as OpenSSH's monitor does. In our implementation, an exploited sthread truly gives the attacker no extra knowledge about users.

Finally, we would like to comment on some benefits that OpenSSH could have gained if implemented using sthreads. There has been a vulnerability in OpenSSH's use of the PAM library, where sensitive data remained in memory due to OpenSSH's failure to scrub memory [35]. This illustrates two points. First, a default-grant strategy is more risky in practice than default-deny, since one must *remember* to scrub sensitive information. Second, it is a common scenario for people to use third-party libraries, and it is very difficult to know all temporary storage of data so that it can all be scrubbed (*e.g.,* fprintf buffers). By using sthreads, PAM authentication would have lived in a separate callgate, and any temporary buffers would have been inaccessible to any other sthreads or callgates, hence avoiding the vulnerability.

Comparing our line counts with those from privilege-separated OpenSSH [57] is not a fair comparison due to the difference in our approach.[1] Most importantly, our OpenSSH version requires no built-in privilege separation mechanism, since it uses sthreads that are implemented in the kernel, and we do not count those in our attack surface. OpenSSH instead counts its privilege separation mechanism as part of the privileged code. Our mechanism is "once for all" so it is more general, whereas the OpenSSH mechanism is specific for their purposes, and perhaps more minimal. That said, OpenSSH's unprivileged code consists of 10,360 lines, and and the privileged authentication code 803 lines, *cf.* 26,114 and 1,550 lines respectively with sthreads. In both our version and in privilege-separated OpenSSH, the authentication (callgate) code is about 6–7%. However, OpenSSH additionally requires 1,700 "miscellaneous" trusted lines, and 900 monitor lines. We instead rely on our 2,000-

[1] The authors of privilege separation [57] seem to recognize that precise line counting is a difficult problem—their caption for line counts reads "source code lines that are *executed*".

line (more general) kernel implementation. We conclude that the trusted code in our OpenSSH is similar to that of privilege-separated OpenSSH, although our overall attack surface, which includes kernel sthread support, is larger (perhaps double) because it is more general. Finally, a benefit of our more general system is that we only had to change 620 lines in OpenSSH to make it more secure.

5.4 Firefox & libPNG

Client-side exploits are as serious as server-side ones, and partitioning client applications is a largely unexplored problem. To illustrate the relevance of client-side exploits, consider an attacker that could exploit a user's browser to obtain the user's credit card information when he next does an online purchase. We therefore need to examine the suitability of sthreads in this context. To experiment with client-side applications and libraries, we isolated Firefox (v2.0.0.1) from libPNG. In other words, vulnerabilities in libPNG will be contained and exploits will not have access to the rest of the browser (such as cookies). We chose to tackle a very narrow and relatively simple problem in order to see how much effort is needed to incrementally secure applications in small pieces, one at a time. Indeed, the coding and design effort were minimal, so we do not discuss this example much, but it illustrates that applying sthreads across a library boundary can be very simple. This is because the interface between a library and applications is well defined, such that cross-boundary data dependencies are usually kept to a minimum. So it often is simple to tag the few memory objects that the library needs from the client.

5.4.1 Threat model

Our goal is to limit the damage of an exploit occurring in the PNG decoding routines. Such an exploit must have no access to Firefox's memory, which contains much sensitive user data. A successful exploit should only be allowed to output an image, which could have been done anyway without requiring an exploit. Hence, the attacker must gain no extra benefit from exploiting libPNG. Of course, we still rely on any callgates being unexploitable.

5.4.2 Design and discussion

To do this, we simply run all libPNG operations in an sthread, giving it permission to output data into the appropriate Firefox image buffer. The sthread can effectively only output RGB image data, which meets our security requirement. The only input to the sthread is the raw PNG data, which is not sensitive, because if an attacker

were trying to exploit the sthread, he would be the one supplying the PNG data (exploit), which is therefore known to him. The only possible attack is to output an arbitrary image, but that could have been achieved anyway without exploiting an sthread by simply supplying the desired image as a PNG. The SELinux policy for the sthread is the default-deny one, which among other things blocks all disk access and IPC.

Our changes to Firefox were rather simple (284 lines) and we noted that many libraries can be wrapped in sthreads in a similar fashion to enhance security. Large applications such as Firefox often were exploited by vulnerabilities in the libraries they use [21], so there is real benefit in enforcing the boundary between the client and library using sthreads. Indeed the libPNG exploit [21] would occur in sthread code and be contained. In fact, any libPNG exploit would be contained as all of libPNG's code runs in an sthread. We also note that we did not use Crowbar to produce this partitioning: the memory dependencies between Firefox and libPNG were trivial and well defined in libPNG's API documentation.

5.5 DNS server written from scratch

We wrote a DNS server from scratch to gain more experience with how it feels when writing code with sthreads, and to highlight the performance cost of sthreads. Remarkably, even though the programming model differs significantly from what we are used to (`fork`), we found that having a default-deny model rather than a default-grant model did not make our programming more difficult. When implementing, the programmer knows what data needs to be shared, so the programmer knows when to use `smalloc` rather than standard `malloc`.

DNS is also a good test of the performance cost of sthreads which we evaluate in Chapter 6. The service is very lightweight (essentially a hash table lookup) and is typically implemented as a single-threaded application, resulting in a fast application since there is no context-switch overhead. To secure DNS, we need to add a relatively expensive sthread overhead. We shall be comparing it to a single-threaded implementation, so the cost of sthreads will be evident and we can assess whether their price is one we can afford to pay.

5.5.1 Threat model

Our security goals for DNS were tailored to explore the functionality of sthreads rather than strictly being real-world requirements one would want. After all, one may argue that DNS holds no sensitive data, so why protect it at all? Integrity is what matters most in this context, and sthreads address this issue as they do

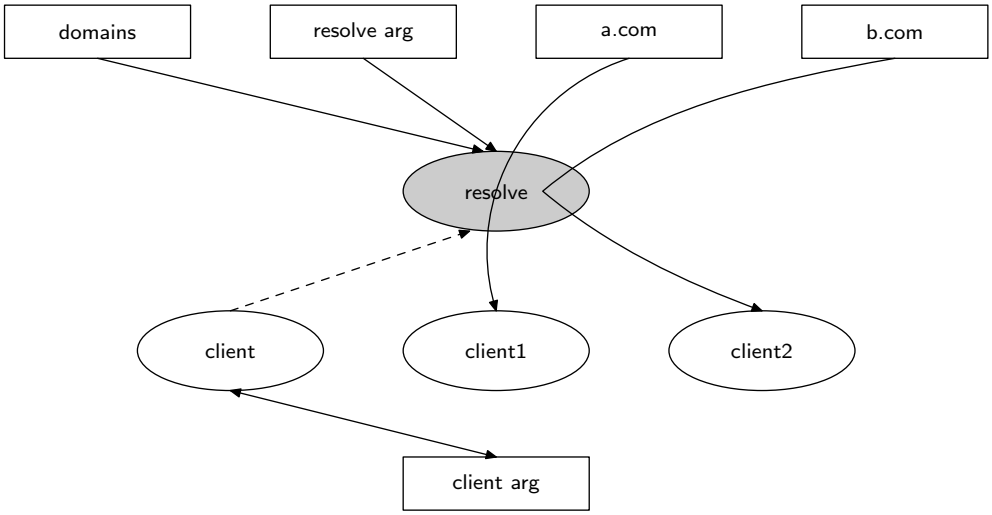


Figure 5.9: DNS partitioning.

with privacy. An exploited DNS server has high impact, since all of its users can arbitrarily be "rerouted" if the attacker spoofs DNS replies. Hence, in a real world situation, one would at least want to isolate attackers from other legitimate users, which we do, although many of today's DNS servers (*e.g.*, Bind) do not. The goals for our DNS server are:

1. Isolate clients. Prevent one client from learning the queries of another one, or injecting responses.

2. Enforce ACLs. Allow clients to access only the zones they are allowed to, based on ACLs and the client's IP address.

3. Deny access to zone data. Disallow a client from obtaining all zone data, while still being able to resolve individual names.

As usual we allow the attacker to exploit sthreads but not callgates. We now describe a design that meets all three goals.

5.5.2 Design

Our DNS server is partitioned as shown in Figure 5.9. To meet the first requirement of isolating clients, we use sthreads. We run each client in a separate sthread, so there is no way that one can read or write data of another one. Note that only the master has access to the UDP socket, and the client sthreads return the data to be written to a particular user. To meet the second requirement of enforcing ACLs, we use per-sthread security policies. DNS ACLs are based on the IP address of

the client making the request. When the master accepts a request, the IP address is known too (from the `recvfrom` call). The master can therefore look up the IP address in the ACLs to determine which zones the client is allowed to access. Note that the master does not parse the DNS query—it only computes over the IP address, which we argue can be sanity-checked, so the master cannot be exploited by a remote attacker despite "reading" (without looking / touching) from a socket. When creating the client handler sthread, the master grants the sthread access to the allowed zones. This is an example of creating an sthread of the same class but with different permissions depending on some other condition (IP address in this case). We specifically constructed our DNS requirements in such a way that we can expose and evaluate the flexibility of sthreads. If the policy were in a static external file, perhaps we would not be able to express such a variable policy where the number of zones and ACLs (and hence sthread classes) are known only at run-time. This is why we chose to embed policies in code, to make their construction dynamic.

To meet the third requirement of giving access to zone data but not allowing its enumeration, we use callgates. This requirement illustrates the problem of needing to read data without giving complete access to it at the same time. Callgates solve this problem well since we can take a hostname as input and return an IP address as output. It is impossible to obtain all the zone data this way. The attacker would have to query all possible hostnames and this could have been accomplished from the network anyway. Rather than giving sthreads access to zone data, we give callgates access to it. We then allow sthreads to invoke the relevant callgate. As with sthread permissions, callgate permissions can be created on a per-sthread basis. Hence, one sthread can be allowed to invoke the resolve callgate which in turn can only read the zone for `a.com`, whereas another sthread could be allowed to invoke the same callgate, although with different permissions—for example, permissions that allow both the `a.com` and `b.com` zones to be read.

Both the callgate and the sthread in our DNS server have the default-deny SELinux policy, giving them no disk access and no IPC capabilities. The sthread does not need network access as the master reads and writes on its behalf. The master reads a buffer from the network and passes it to the sthread opaquely. Similarly, the sthread returns an opaque buffer which the master sends to the network. This prevents the sthread from taking over the UDP/53 socket to serve arbitrary clients. By managing the buffers opaquely, the master is unlikely to get exploited from any malicious contents in the buffer.

Component	Line count	Percentage
Total	1,185	100%
Sthread	1,131	95%
Callgates	54	5%

Table 5.9: DNS line count.

```
char *resolve(char *domain, int hash);
```

Figure 5.10: DNS callgate interface.

5.5.3 Avenues for exploitation

We briefly discuss the attack surface of our simple DNS server. Note that our implementation is minimal but complete enough for performance evaluation and analysis. For example, we only support 'A' records and not all DNS records and queries (*e.g.,* reverse, MX). Table 5.9 shows line counts for sthreads and callgates. We note that a small fraction of code lies in callgates, and we expect this to remain so in a complete implementation, since the only functionality required essentially is a hash table lookup (*e.g.,* hostname to IP address). More important, though, is the rather minimal interface of callgates, as shown in Figure 5.10. The `resolve` callgate takes two parameters: the domain name to look up and its hash. Since we are forcing the client to compute the hash, instead of leaving it to the callgate, we do not need to trust the hashing code. The client must still supply the domain name in case of collisions, where a linked list is used. The attacker has little leverage for exploiting callgates since they only take a single string as input. Callgates can definitely sanity-check input ensuring that it is composed of acceptable characters and length, therefore reducing the risk of exploits. Of course a complete implementation would require more parameters such as the type of record being looked up. For now it is hardcoded to a DNS A record, but adding an integer parameter for the type of record does not give an attacker much extra leverage. It is very difficult to cause a full blown exploit with the use of an integer which can be limited to only a few values (*i.e.,* DNS query types).

5.6 Coverage provided by tools

So far we discussed how we applied sthreads to applications. We now turn to Crowbar, our tool that helps partition legacy code, and how useful it was during development. First, though, we assess based on our experience Crowbar's important

requirement of high code coverage when gathering traces. Being a run-time analysis tool, Crowbar only gives information on past runs and does not guarantee that future runs, which may be different, will work. This leads to the question: how brittle is an application that is partitioned by using only Crowbar without any other manual work? We have two test cases for answering this: Apache and OpenSSH. In Apache's case, a single Crowbar run was enough for obtaining all the information necessary for making all our future runs succeed. None of our tests or benchmarks failed due to insufficient privilege.

In OpenSSH's case, we were required to produce a trace for each different authentication mechanism. This did not come as a surprise to us, since we were required to do so anyway in order to identify the different callgates, and permissions for the different authentication mechanisms. We did, however, experience an unexpected crash with OpenSSH when connecting from a different test host. In our prototype, we only support DSA for key authentication, leaving behind RSA. One particular client machine we were using had only RSA enabled, causing the client session to crash due to insufficient memory permissions.

We tested some exceptional cases with OpenSSH such as authentication failures, timeouts, and passing on to the next authentication mechanism. They all worked as expected. This leads us to believe that an adequate test suite for obtaining traces provides sufficient coverage for adding sthread support to applications, and expecting applications to run robustly, without crashing. Our experience shows that crashes due to insufficient memory permissions occur when doing actions completely different from those when obtaining traces (*e.g.,* different authentication mechanisms). This typically occurs when the sthread needs a new capability in order to perform some new functionality.

5.7 Assistance provided by tools

We now relate how much Crowbar helped us in partitioning existing code. Our two main development efforts on existing code are Apache and OpenSSH. In both cases, we actually produced two partitionings, one before Crowbar was available and one afterwards. Although we have no objective scientific evidence, we definitely believe that Crowbar was an immense aid for us the second time we secured the applications. For what it is worth, it took almost one month to partition Apache without Crowbar, and just a couple of days the second time round using Crowbar. Of course we were more familiar with the code the second time round, and roughly knew where to look, although Crowbar did spot and remind us of all the necessary changes, some of which we even forgot about.

We attempt to evaluate the usefulness of Crowbar in a more objective manner by examining its output. In OpenSSH's case, Crowbar indicates 6 files out of the total 120. This gives a developer a much more concentrated subset of source files to study rather than having nowhere to start and simply following code, which could lead to examining dozens of files. We also note that in some cases, for example with global variables that have obvious and self-describing names (*e.g.,* `debug_level`), one does not even need to understand or follow the files pointed out by Crowbar. The tool reports information at a line granularity so one can merely open the file pointed out, go to the appropriate line and (say) change a `malloc` to an `smalloc` or wrap a function in a callgate.

It is important to remember that Crowbar is also useful when dealing with code written from scratch. Suppose that we implemented an SSL web-server from scratch that isolated users, but without considering the man-in-the-middle attack. To add the extra protection we likely need to change our partitioning quite a lot (*e.g.,* have two phases) and things may break due to insufficient privilege. Even though we have no real direct experience with doing large refactoring on sthread-enabled code, we do believe that Crowbar's permissive mode will help immensely in such cases, just as it helps when adding sthreads to new code. After all, the two problems are rather similar in nature—refactoring is modifying existing code, and we know Crowbar helps there.

5.8 sthreads and Crowbar: benefits and drawbacks

Now that we used sthreads and Crowbar in practice, we look back and comment on our original design choices. We feel that when implementing code running in an sthread, the programmer has greater peace of mind and need not be as paranoid about all those sanity-checks needed on user input.

Partitioning existing code is difficult. Our APIs geared for legacy code (*e.g.,* `smalloc_on`) are not mere syntactic sugar and aids, but are instead quite fundamental. Without them, it is a lost battle. The same goes for Crowbar. We feel that these are necessary. Of course one can do without them but the effort is enormous (we tried). For this reason we continue stressing that a key part of the problem is making partitioning work for legacy code and that our mechanisms for doing so are very important and not just handy bonuses. With Crowbar, partitioning legacy code becomes turning the crank—the way it should be. All the thought goes into design and not into implementation details and realization. We refrain from having a fully automatic solution (Crowbar could attempt) because we still believe that thought should go into defining the threat model and solution design. Consider our parti-

tioning against the man-in-the-middle attack: it would be difficult for a machine to spot the problem, and even more so to develop a solution. Overall, we feel that sthreads are an appropriate tool for securing new applications, and the combination of helper APIs and Crowbar enable the (easy) use of sthreads on legacy code.

Is the combination of SELinux and sthreads a good one or would have sthreads benefited from their own system call protection mechanism? SELinux provides a good way for specifying policies at a "high level" rather than focusing on individual system calls. For example, one can allow a file to be "read", without having to explicitly allow `open`, or `mmap` or all the possible ways of obtaining a file descriptor and reading from it. Thus SELinux provides a relatively simple (and powerful) way of controlling filesystem accesses. It is adequate for network access and other forms of IPC, too. Since SELinux is relatively "high level" it cannot be used for fine-grained blocking of individual system calls. For example, one cannot deny the use of `gettimeofday` or `getpid`, even if unused by an sthread. Thus, for a truly default-deny implementation from a system call point of view, sthread would need a mechanism (even a simple bitmap) to block specific system calls. This system call firewall could be used in conjunction with SELinux. In practice, however, we found that SELinux's protection is strong enough to contain attacks. The most useful thing that an attacker can leak despite SELinux's protection is likely the time of day.

5.9 Summary

Our SSL web server implementations show that sthreads are powerful enough to tackle complex threat models such as an attacker being able to exploit the server and act as a man-in-the-middle: a scenario which has not been considered and tackled to date. Having applied sthreads to Apache & OpenSSL, OpenSSH and Firefox & libPNG, we conclude that it is practical to apply sthreads to real and complex applications, especially with the aid of Crowbar which significantly reduces the amount of code that needs to be studied and indicates where changes need to be made. We never had to change more than 1,700 lines in existing applications. We also wrote sthread-enabled applications from scratch (DNS server, web server) and did not encounter any difficulties despite sthreads having a programming model (default-deny) opposite to that of standard processes and pthreads.

Chapter 6

Performance

We start our performance evaluation with microbenchmarks. We focus on sthread creation/destruction time as it is the main overhead, and we show the benefits of "recycling" sthreads, *i.e.,* safely reusing them from a pool. We then discuss the end-to-end performance we achieve when applying sthreads to real applications. We examine a mix of newly written applications, existing ones partitioned with sthreads, servers and clients, to gain a broad view of performance. We partition existing applications without paying attention to memory use, allowing us to measure the performance overhead of sthreads when used in a straightforward way, without optimization attempts. In our high-throughput server applications, we compare our kernel sthread implementation to our userspace one to examine the additional cost of the latter in real scenarios.

We run all benchmarks on a single-core 2.66GHz Intel Xeon with 4GB of RAM, running Linux 2.6.28, with SELinux in enforcing mode and all security mechanisms enabled. We use a single-core machine to ease saturation of the server's CPU when benchmarking server throughput. While we have not investigated the performance of sthreads on multi-core CPUs, we do not anticipate that the introduction of sthreads when partitioning an application will change the application's scaling behavior on multi-core CPUs significantly. We note that sthreads are well suited to multi-threaded applications and can be used in those contexts, so we expect that sthreads will be able to leverage multi-core systems effectively for higher performance.

We conclude by examining the performance of Crowbar, our tool that developers will use to partition existing legacy code. We discuss the latency the tool brings to the development cycle of applications.

Figure 6.1: The context switch overhead of processes is 12% greater than that of threads.

6.1 Microbenchmarks

So what are the performance implications of using `fork` (*i.e.,* processes) to implement sthreads? We examine two costs: context switch and process creation overhead. We discuss other sthread-related creation costs, such as zeroing the stack, restoring globals, and memory usage later in this chapter.

Processes incur a context switch overhead because each requires a different page tables. Loading a new page table causes a TLB flush, so memory accesses in the immediate future (as the TLB fills) will be slower. This does not occur with standard threads, as they share the same page table. On Linux, threads are implemented in the kernel, so all other aspects of context switching (compared to processes) remain the same. That is, we still pay the price of entering the kernel and saving and restoring registers there. We therefore expect the context switch overhead of processes to be greater than that of threads, and this extra cost should come from TLB misses. Figure 6.1 shows the context switch latency of processes versus pthreads. The benchmark consists of two long-lived processes (or threads) aggressively switching back and forth, using a system call we added to force the switch immediately.[1] We measure the time between one process executing (in userspace), until the next process starts executing (again, in userspace). By reading the PAGE_WALK hardware performance counter using perfmon2 [53], we know how many cycles were spent by the CPU walking the page table due to a TLB miss. We divide this by the total cycle count of the benchmark to obtain the percentage of time wasted due to TLB misses, as recommended by the Intel performance manual [26]. The `fork` benchmark spent 7.7% of the time page walking, accounting for 75% of the difference from the `pthread` benchmark. The extra cost comes from the greater number of

[1] An alternative is to force switching using a semaphore or pipe, though this incurs more work per thread, amortizing some of the switch cost.

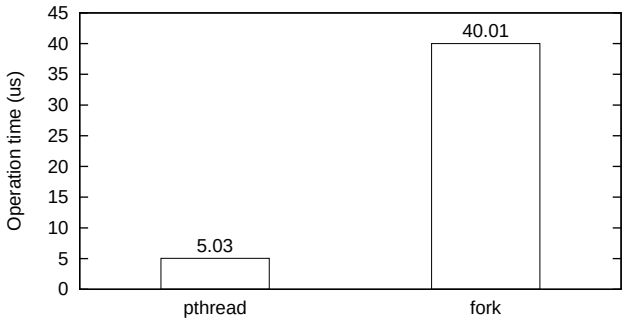


Figure 6.2: Process creation overhead is 8 times greater than that of threads.

instructions involved when switching processes, some of which are serializing, such as reloading the CR3 register. The lesson from this benchmark is that pthreads context switch about 12% faster, so this approximately is the gain we can expect if we choose to optimize the sthread implementation's context switch overhead. If we add computation to each thread, we expect the win to decrease, as more time is spent in useful thread execution rather than in switching overhead. We conclude that the context-switch overhead of processes is not a huge penalty.

But how fast can we create and destroy processes? If we create a new sthread to handle each individual user of a network server, then the sthread creation and destruction overhead could limit the throughput of requests that we can serve. On Linux, creating either a pthread or a process follows the same code path in the kernel, though the significant difference is that, in the case of `fork`, the memory map of the parent is copied to the child, rather than shared. Thus, we expect pthreads to be faster to create, because this copy is avoided. Figure 6.2 shows the latency of creating and destroying a thread and a process. That is, we measure the total time it takes to `fork`, `exit` the child, and `wait` for the child. For pthreads, we use `pthread_create`, `pthread_exit` and `pthread_join` instead. As expected, `fork` is slower, though it is *much* slower. This is due to the nature of this benchmark and the details of how the page table is copied. The benchmark consists of a short-lived process that exits immediately, and `fork` is particularly bad at handling that case. The following assumptions were made in Linux's `fork` implementation, and hence the following phenomena occur in this benchmark:

1. The page table is not copied completely. It is populated at run-time, on demand, as page faults occur, as memory is accessed. The page table is essentially demand-loaded, so only those entries that are used are populated. This is good because the price of `fork` is cheap, and we pay a setup price only for pages actually used. Long-lived processes amortize the cost of page-faults required

for this demand-loading. For short-lived processes, we incur extra page faults as we commence executing, but in our benchmark we exit immediately after that, so page fault overhead dominates.

2. The pages are marked copy-on-write (COW) in both the parent and child. This has two main effects. First, any write that the child performs will incur a page fault and a copy of the page. This highlights the cost of short-lived processes that write to the stack (as `fork` returns) and then exit immediately. Second, even if the child exits, the parent's pages are still COW. If the parent writes, it will still get a page fault even though no copy is performed. No copy is necessary as the child died and the page is now unshared, but we still pay the price of the fault.

We obtained a profile of our `fork` benchmark by using OProfile [3], a system-wide (includes kernel) sample-based profiling tool for Linux. As expected, the profile reveals that 90% of the time is spent page faulting, for the reasons we discussed above. `fork` performs well for long-lived processes, after the page table "warms up", though for short-lived ones, its page-table optimizations hinder execution speed. For sthreads, we need something that performs well in both cases.

6.1.1 Recycling sthreads

So how can we improve the rate at which we can `fork`, execute, and `exit`, in the context of sthreads? Rather than creating and destroying processes each time, we can keep a pool of long-lived processes and reuse them as necessary. This is a standard technique used, for example, in high-throughput servers (such as the Apache web server). Since these processes are long-lived, the page faults incurred by `fork` will happen only initially, and subsequent runs will be faster. To implement such reusable processes, we must first create a child using `fork`, after which that child will sleep until it is instructed to be reused. The steps then become:

1. Parent signals the child to start.
2. Child executes.
3. Child signals completion to parent, and sleeps.
4. Parent waits for child's completion.

Note that these operations can be ordered differently depending on scheduling, though this particular one is the fastest as it requires fewest context switches, and so we pick this for discussing the best that we can do. The cost involves two context

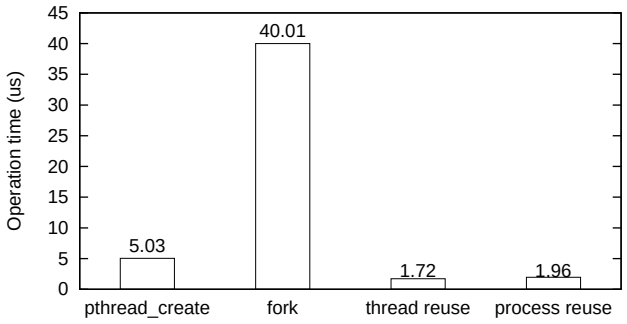


Figure 6.3: By reusing processes rather than creating new ones each time, we execute 20 times faster.

switches (between steps 1–2 and 3–4) and three system calls (steps 1, 3, 4). By reusing processes from a pool, we no longer pay the high process creation cost (except the first time), but only pay context-switch cost, which we have already shown to be competitive with that for (kernel) threads. Figure 6.3 shows the cost of process and thread creation compared to the cost of reusing processes or threads from a pool. We measure the time it takes to create and destroy a process or a thread, as before. In case of reuse, we measure the time it takes to signal a child to start, and wait for its completion. To signal and wait, we added system calls specifically for these purposes that outperform generic (and more complex) semaphore mechanisms. (We do not use standard signals as they cannot be used to stop and start individual pthreads, and we wanted to evaluate pthreads too.) As expected, reusing processes is much cheaper (20x) than creating them anew, since there are no page-fault costs. The dominant cost is now context-switch time, which is not much larger than that for pthreads.

But what are the security implications of reusing processes? After a process executes, it will have written to its stack and heap, and such data may be sensitive if, for example, the process was handling a credit card transaction. Upon process reuse, the next client, if malicious, could attempt to recover such data. In the context of sthreads, we therefore have more work to do when reusing processes. We need to scrub sensitive data, and restore any initialized variables (*e.g.,* globals) to their original value. We call this mechanism recycling sthreads. The cost of recycling an sthread is proportional to the amount of memory it writes to, and how much of it is COW. In the latter case we pay the price of a `memcpy` and in other cases the price of `memset` zero. Of course we still incur the context-switch overhead associated with starting and stopping the sthread. Figure 6.4 shows the invocation cost of reusing from a pool of threads, processes, and sthreads. We measure the time it takes to invoke a child, make it exit, and be notified of completion. Standard

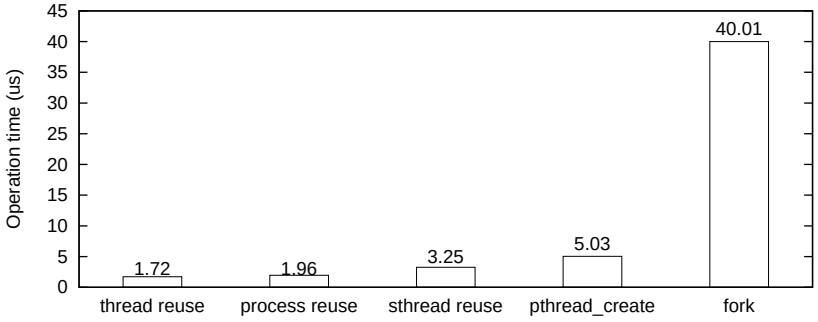


Figure 6.4: Recycling sthreads is 12 times faster than creating them anew with `fork`.

Time (cycles)	tot. %	diff	diff. %	Event
0	0.0	0	0.0	`sthread_create` starts
2832	10.8	2832	10.8	parent enters `restore` syscall
8048	30.7	5216	19.9	child wakes up from `deep_sleep`
9472	36.1	1424	5.4	child executes
10560	40.3	1088	4.2	child enter `deep_sleep`
11800	45.0	1240	4.7	child starts recycling
14952	57.1	3152	12.0	did page walk and `memset` zero
17048	65.0	2096	8.0	did `memset` zero
18992	72.5	1944	7.4	did `memset` zero
20136	76.8	1144	4.4	finished recycling
23384	89.2	3248	12.4	parent's `sthread_create` returns
24640	94.0	1256	4.8	parent entered `sthread_join`
26208	100.0	1568	6.0	`sthread_join returns`

Table 6.1: Breakdown of sthread recycling cost. Recycling dominates and is variable.

pthreads are fastest and represent the baseline for the three system calls and two context switches involved. There is of course no memory protection in this case. Processes buy us integrity (when not reused) since memory writes are isolated across processes, though the cost of processes is greater because they run with different page tables, so extra registers must be restored (*e.g.,* CR3 for the page-table), but most importantly, the TLB is flushed, causing future memory accesses to be slower. Recycling sthreads additionally buys privacy at the increased cost of `memset`ting sensitive data to zero, and `memcpy`ing any COWed data back to its original contents hence increasing integrity even further. The cost of creating fresh threads and processes are shown in the figure for reference, to highlight the improvement of reuse.

We now discuss the precise cost of recycling an sthread, and assess how much room for improvement there is. We do so by examining the detailed breakdown of execution time for our sthread recycling benchmark. Our benchmark consists

of calling `sthread_create` immediately followed by `sthread_join`, and the child sthread exits immediately. Note that we run the benchmark code twice so that the second time the new sthread is created by recycling, and it is this second run we measure. This benchmark therefore captures the time it takes to recycle an sthread, start it, and finish it immediately. We instrument the userspace code by recording the CPU cycle counter at various points, and do the same for certain kernel code paths using kprobes [16], a lightweight Linux mechanism for dynamically adding instrumentation code to the kernel.

Table 6.1 shows a detailed breakdown of the execution time within our sthread recycling benchmark. The rows of the table represent actions in order of execution. The first column is the absolute time measured in cycles, and the second is the percentage of total benchmark time elapsed so far. The third column represents the time it takes for the action in that row to occur, and the fourth column expresses this as a percentage of the total benchmark time. The fifth column describes the event. We now discuss the results in detail.

First, we spend 10% of the time before we actually enter the kernel to perform the `restore` system call. The userspace library needs to determine whether there is an sthread available for recycling, and this involves comparing the security context of existing sleeping sthreads in the recycle pool with the one supplied in the call. There likely is room for optimization here, though we did not investigate doing so since the main bottleneck is the total recycling (*i.e.,* all `memcpy`s) and context-switch time. The kernel then needs to wake up the sleeping sthread. Note that we do not need to recycle at this point, since we recycle upon exit. Waking up the child involves context switching (hence flushing the TLB) and activating Linux's scheduler, costing us 20% of the time.

The sthread then executes for 5% of the time, and manages to enter the kernel (to exit) in 4% of the time. The latter is a good estimate of how long it takes to enter the kernel sthread code. That is, the time of executing the interrupt 0x80 instruction, the delivery of the hardware interrupt, the context-switch overhead to the kernel, the time for executing the generic kernel interrupt handler, and the time it takes to reach our sthread system call handling code.

We then need to walk the page table in order to determine what memory to clear and restore. We perform the page table walking and clearing together in one pass. For this benchmark, we had to `memset` zero three stack pages, costing us about 8% each. The first `memset` is more expensive because it includes the time spent walking the page table until we found an entry to clear. Subsequent `memset`s were of adjacent pages, so there was negligible walking left to do in order to find them. An alternative implementation could choose to scrub only the first couple of stack

pages, and unmap the rest to amortize `restore` time.

We then context-switch back to the parent to indicate that sthread creation completed, though in this case the sthread even had time to exit too. This context switch costs us about 12% of the total. We then enter the kernel again to join the sthread (*i.e.,* wait for its completion).

In the previous benchmark, sthreads were zeroing only three pages (stack). Figure 6.5 shows how the cost of sthread creation varies as the number of pages an sthread modifies increases. We increase the number of stack pages and pages containing globals (marked COW) that the program writes to, and measure sthread recycling time. To clear stack memory, we need to `memset` to zero, whereas to restore globals, we need to `memcpy`. The curve labeled "memcpy & memset" represents a run where the same number of stack and global pages are touched. Since this curve deviates relatively little from the `memcpy` curve, as expected, the `memcpy` cost dominates. The cost increases more with COWed data as `memcpy` is more expensive than `memset`. COWed data is mainly used to store globals, so when designing new applications for sthreads, writing to globals should be kept to a minimum, or the globals should at least be grouped in as few pages as possible. It is worth noting that statically compiled binaries perform better for this reason, since globals are lumped together in fewer pages, rather than scattered throughout the address space; also, the absence of a dynamic linker will not modify GOT & PLT entries, and thus not cause writes to COWed areas. From this benchmark, we conclude that sthread creation performs much better when an sthread writes to a small number of pages containing globals.

Note that if globals are checkpointed when the program starts, *i.e.,* when `main` is called, their contents is often zero (uninitialized globals, in `.bss`). With sthreads, we do in fact checkpoint when the program starts, prior to the main program's execution, in order to avoid checkpointing any sensitive data, so this observation is relevant. In our benchmark, for example, of the twenty pages containing globals (including those of libraries), only five were non-zero. Indeed, only 2,722 bytes were non-zero. Thus, an optimized version of `restore` could `memset` zero rather than `memcpy` zero where appropriate.

6.1.2 Callgate optimizations

We now discuss other optimizations we applied to our system, relating to callgates. One of our assumptions is that callgates remain unexploited. If this property holds, there is no need to create a new callgate on each request and scrub its memory when it is done. Because the callgate is trusted, and we assume that it will not be exploited,

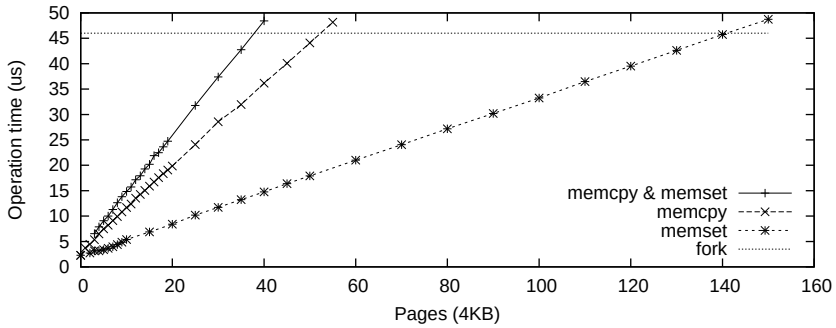


Figure 6.5: Sthread recycling cost versus amount of memory written to. Writing to COW mappings degrades performance most.

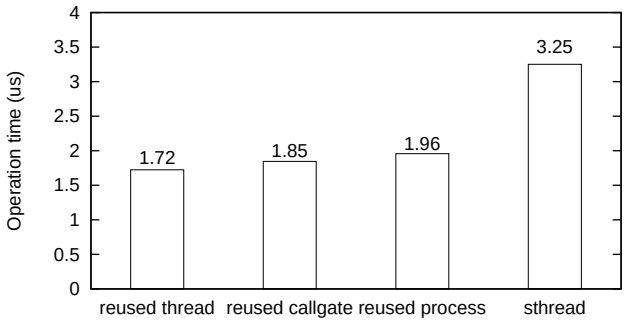


Figure 6.6: Reusing callgates is almost twice as fast as creating new ones each time.

any sensitive memory it holds will be kept secret. Thus, to improve performance, we can employ long lived callgates that are reused each time. This is different from *recycled* sthreads that are scrubbed on reuse—with *reused* callgates, we just reuse them as they are. Hence, callgate invocation now merely costs context-switch time, as shown in Figure 6.6. The security price to pay is that callgates are now shared across sthreads, so if a callgate is exploited, the attacker could compromise other sthreads by returning exploits from the callgate. If the callgate is critical to the application, though, as is an authentication callgate for OpenSSH, then this additional risk is minimal—if the attacker can authenticate as root, then all bets are off anyway.

The process and thread bars are for reference to compare against reusing these resources. Reused threads are faster because there is less context-switch overhead (due to TLB misses) as compared with for processes. Reused callgates outperform reused processes because they invoke fewer system calls. With processes, we need a total of three system calls: first we signal the child to start; second, the child signals termination; and third, the parent waits for the child's termination. For callgates, we can combine the first and third calls, because the execution of the caller must

block until the callgate completes. Hence, callgates require a single invocation of a system call that will signal the callgate to start and wait for it to complete, all in one call. This approach is not applicable for processes, since a master process coordinating many children may wish to continue to run (so it can manage children) rather than block until child completion.

6.1.3 Tagged memory optimizations

Our final optimizations concern tagged memory. To create and destroy tagged memory arenas, we use `mmap` and `munmap`, respectively. This has a system call overhead, the kernel must set up the virtual memory areas in the process control block, and page faults occur to populate the memory areas when accessed. We avoid this cost by keeping a free-list of arenas in userspace and zeroing memory upon reuse. To ensure that memory is indeed scrubbed, the parent sthread is responsible for zeroing memory handed over to child sthreads. For example, a parent sthread could create a tagged memory region to pass arguments to a child sthread. When the child sthread exits, the parent can recycle the tagged memory arena for future use by zeroing it. This has no security implications because the parent is trusted with respect to its children.

Figure 6.7 shows the benefit from using a free-list in userspace. This optimization pays the greatest dividend when the tagged memory region is small, and hence the common cost of `memset` dominates less, so the overhead of system calls and page faults present only in the `mmap` case is relatively greater. As the memory region's size increases, both implementations become bottlenecked by `memset`. Programmers seeking maximum performance must take care to pass relatively small read-write arguments to sthreads so that recycling time remains short. To improve performance in the case of large tagged memory regions, one optimization would be for the kernel to recycle the arena and zero only the memory that was actually written to (*e.g.,* by inspecting whether the page is read-write), much as we do when we restore COW mappings. This way, we pay the price only when we need to, without assuming that all pages were written to.

6.1.4 Userspace implementation

The key performance differences with respect to the kernel implementation are:

1. We cannot be smart about COW memory segments as we lack page table information. In the kernel, we could check whether a page was still read-only (COW), do no work for that case, and do a `memcpy` otherwise. In userspace, we are forced to assume the worst case and remap everything, which causes old

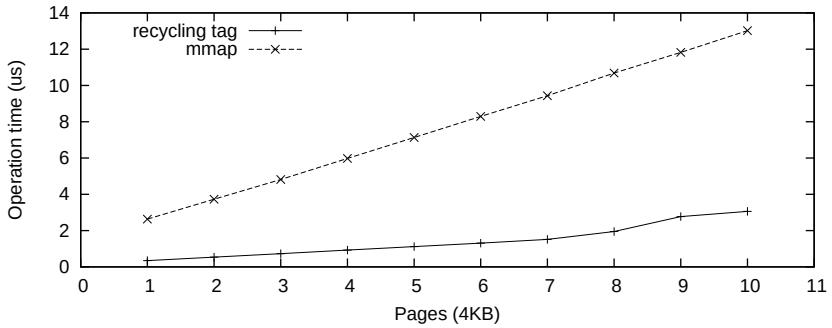


Figure 6.7: Tag recycling time versus arena size.

pages to be reclaimed and new mappings to be set up, and may cause future page faults on write.

2. Updating file descriptors over UNIX sockets is slow. This is a major drawback, as passing sockets to sthreads is common when implementing network servers.

3. Many system calls for one logical operation. For each memory segment, we need a system call, rather than doing it all in one lump. Multiple calls to `ptrace` are required, too—to stop the process, change EIP, and restart it. In the kernel, we can do it all in one go, avoiding the cost of entering the kernel multiple times.

We therefore expect the userspace implementation to be slower, though we note that when dealing with non-COW memory, we expect it to be efficient—checking System V shared memory statistics is fast. COW support is needed chiefly for legacy applications, so for newly written code, the userspace implementation may perform adequately. Unfortunately, though, for file descriptor passing, we have no faster solution than using UNIX sockets. Figure 6.8 shows the cost of recycling an sthread in our userspace implementation compared to in our kernel one. The overhead for passing a file descriptor is most notable, and the `mmap` required for COW memory is expensive. The "sub sthreads" column consists of sthreads of sthreads. This is a particularly difficult case for our userspace implementation as extra IPC is required to create grandchild sthreads, to preserve security semantics. Such sthreads are twice as slow than direct sthreads created by the master.

Table 6.2 details the cost of sthread creation. Our baseline of starting an sthread, stopping the child upon completion, and waiting for the child costs us 34%. This consists of context-switch and system call overhead and there is little room for optimization because we must do this work. Contrary to our kernel implementation, we then pay the price of verifying read-only mappings (6%), and restoring registers

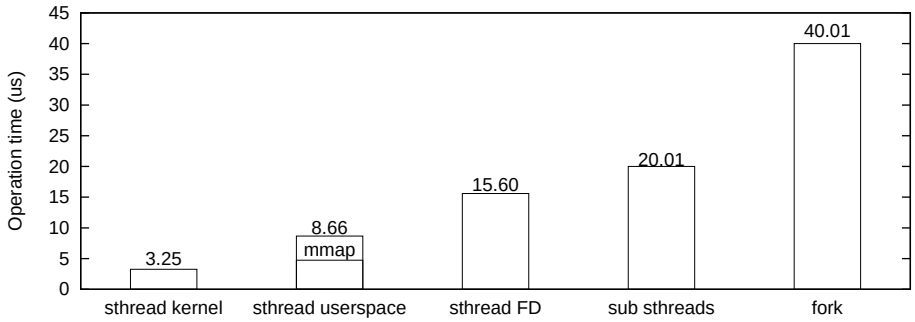


Figure 6.8: Comparison of recycling costs: userspace *vs.* kernel implementation.

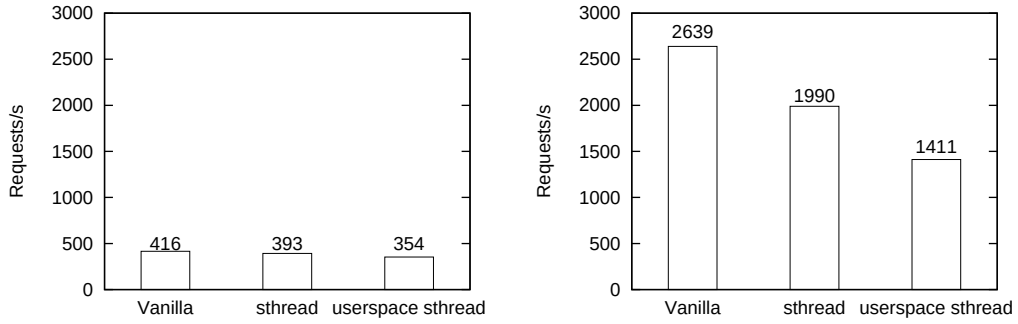
Time (us)	% of total	Operation
2.97	34%	Baseline: start child, child stops, parent waits.
0.50	6%	`shmctl` to check read-only mappings.
0.31	3%	`ptrace` to change EIP.
0.59	7%	two `munmap` to remove extra heap and stack.
4.30	50%	`mmap` to restore globals.

Table 6.2: Userspace implementation sthread creation cost breakdown.

via a system call (3%). The huge price difference, though, is due to remapping globals (50%). In our kernel version, we keep the mapping but just fix the contents of pages, where modified. In our userspace version we really do two things at once: remove the previous mapping, and create a new one, which involves many more kernel and page table operations. Despite the convoluted approach we use in our userspace implementation, it is still remarkably fast compared to standard `fork`, while also providing more protection.

6.2 SSL Apache

We start by examining unmodified Apache v1.3.19 and OpenSSL v0.9.6, running with Apache's default configuration. We refer to this as vanilla SSL Apache. We compare vanilla SSL Apache to SSL Apache partitioned with sthreads. For a single client, our version of Apache creates two sthreads and uses three callgates. This is in contrast with vanilla Apache which handles each client entirely in a single process. We measure the throughput in requests per second for retrieving a single static HTML file. The page retrieved is Apache's default welcome message, a file of 2,673 bytes. We perform the experiment on a LAN, and are CPU-limited. We tune Apache's MaxClients parameter to the value that yields most throughput in our setup. This parameter dictates the number of concurrent processes. Specifically, we



(a) No sessions cached.

(b) All sessions cached.

Figure 6.9: Apache's performance.

set it to 15 for vanilla Apache, and 5 for sthread-partitioned Apache. We examine different settings of this parameter in a later experiment. The workload consists of 200 long-lived parallel client processes, each client requesting serially with no delay between its successive requests. The client processes all run on a single host with four 2.66GHz cores. We examine both with and without SSL session caching.

Figure 6.9 shows the throughput of sthread-partitioned Apache and the original version. There are two cases in this experiment: one where all SSL sessions are cached, and no public-key cryptography is performed, and another one where no SSL sessions are cached, so public-key cryptography dominates the cost. These cases represent the same user connecting over and over again, and new users connecting each time. Typical workloads lie somewhere in between. Figure 6.9(a) shows throughput when no sessions are cached. As public-key cryptography costs dominate, sthreads affect performance little. In this case, our increased security comes at a cost of being 6% slower than vanilla Apache. Vanilla Apache is still faster because it does not need to scrub any of the memory of its workers—it simply reuses them. In the case where all sessions are cached, the cost of sthreads is more evident, as sthread overhead consists of a larger share of the workload. In this case, we perform 25% slower than vanilla Apache. Thus, in a real scenario where there is a mix of new users (no cached SSL sessions) and old ones (SSL sessions cached), we expect our slowdown to be approximately 6%–25%.

The main cost when comparing vanilla Apache and sthread Apache is the extra cost sthreads pay for recycling: *i.e.,* scrubbing memory. To serve one client, the kernel must zero 13 stack pages and restore 25 pages worth of globals. Furthermore, there are 22 pages required for passing arguments between sthreads and callgates, and these must be cleared by the userspace library. Thus, for each client, our sthread implementation requires a `memset` zero of 35 pages, and a `memcpy` of 25 pages. This

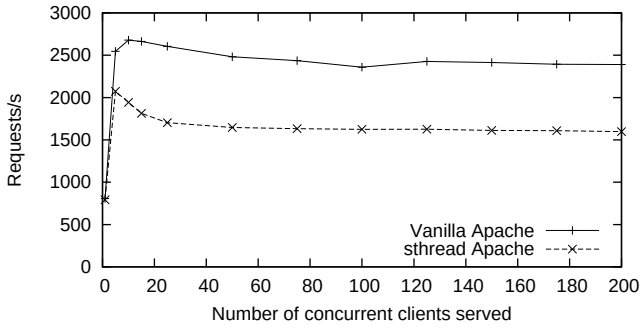


Figure 6.10: Apache's throughput as process pool size (MaxClients) increases.

does not account for any heap usage while serving clients, which adds extra cost, as it requires unmapping the extra memory and zeroing any used pages.

Our userspace implementation is 10% slower than our kernel implementation when cryptography cost dominates (*i.e.,* when no sessions are cached), though it is 29% slower when all sessions are cached. Note that this is a particularly difficult case for our userspace implementation, since our Apache implementation uses multiple levels of sthreads, *i.e.,* sthreads create child sthreads which create even more children. In such cases, there is extra IPC required by the userspace implementation since, for security reasons, only the master is allowed to create sthreads. We have multiple levels of sthreads because we create a "sub-master" that coordinates all the child sthreads and callgates required to handle each user. The main master is only responsible for creating new sthread worker groups, as with vanilla Apache. Figure 6.10 shows how the performance of Apache scales as the number of concurrent clients being served increases. That is, we vary Apache's MaxClients parameter, which regulates how many processes are created, and therefore how many clients can be served in parallel. MaxClients does not cap the *total* number of sthreads created, but rather the number of clients being served. Because we serve clients with multiple sthreads, a MaxClients setting of one does not imply a total sthread count of one. With vanilla Apache, however, there is a one-to-one mapping between concurrent clients and processes. This experiment will therefore show whether the greater number of sthreads per client causes excessive context-switching, slowing down the request rate. Regardless of the MaxClients setting, we keep a constant workload of 200 parallel clients performing requests.

Both vanilla Apache and sthread-partitioned Apache level off, despite the latter creating five times more sthreads / processes than the former. Even though we end up having 1,000 sthreads when the maximum number of concurrent clients is set to 200, the system still continues to serve at a steady rate. This is because most

of these sthreads are sleeping. Even though we have five sthreads per client, only one is active at any one time, as the workload is spread as a pipeline. For example, one sthread starts the handshake and then hands over the connection to the second-phase sthread—they never require concurrent work. The peak throughput for the sthread version occurs at around 5 clients, whereas for vanilla Apache around it is 15. This factor of three difference reflects that we recycle three short-lived sthreads per client; the other two sthreads involved in the client handshake are long-lived callgates for that "worker-group". Thus, when tuning this parameter with sthreads, one must consider how many sthreads are recycled per client.

6.2.1 Measuring memory usage

We now comment on how memory use scales. Given that many sthreads are required to serve a single client, we want to assure that having a large pool of sthreads does not exhaust physical memory. We measure memory use in two ways:

1. **Per-client usage.** We measure how much physical memory is used to serve a single client. This measurement not only reports how much memory is used, but also what memory is being used for, allowing us to pin-point where the overhead comes from. The per-client memory use also allows us to approximate how much total memory consumption to expect when serving a given number of concurrent clients. We do not account for any kernel memory in this measurement, so this metric alone is insufficient because sthreads consume such memory for process control blocks and page tables.

2. **System-wide usage.** We measure the total end-to-end physical memory usage of the system. This accounts for both userspace and kernel memory. This measurement is the "true" physical memory use experienced when running applications, and it also acts as a sanity-check for any estimations based on our first metric. Being a coarse-grained measurement, it does not reveal what memory is being used for, which explains the necessity for the first measurement.

We now discuss how we perform these two measurements. Memory usage changes as the application's state changes. To compare fairly between different runs of the same application, we need to measure memory use when both runs reach the same state. To do this more accurately, we measure memory use *after* a client request, when the application is dormant, and its memory map is "stable" (not changing). This still accounts for any used heap, since even though an application calls `free`, it does not physically deallocate pages by calling `munmap`.

To obtain our second metric (system-wide usage), we use the `free` command, which reports total physical memory use, and how much of this memory is used for disk buffers and other cached objects. This information is obtained by querying, via `/proc`, the kernel's global memory use statistics. Many operating systems, including Linux, attempt to use *all* physical memory by caching files in RAM, and this memory can be reclaimed at any time by writing any changed files back to disk. To get an accurate measurement of the *effective* physical memory used, we subtract the memory used for disk buffers and cached objects from the total memory use.

Regarding our first metric (per-client usage), measuring memory use precisely at a fine-granularity is not trivial. Shared memory and COW make the problem difficult because we need to identify exactly how much memory is shared, and how much of it is resident in RAM. To obtain the physical memory used by a process alone, one could subtract the process's shared memory size from its resident memory size (RSS). The intuition is that if a process attaches to a shared memory segment, it does not increase physical memory usage, because another process already allocated that memory, and so this memory should not be accounted in the per-process memory overhead. Subtracting shared memory size from RSS may be misleading, however, because we do not know whether other processes are actually using that shared memory, and we do not even know whether that shared memory is resident in memory or merely backed by disk. To get a more precise measurement, we use `exmap` [8], which inspects page tables of processes, returning fine-grained information about how memory is used. The metric we found most applicable to measure memory usage is writable memory. Writable memory accounts for the following:

- Any private memory used by the process. If a process allocates memory and uses it, writable memory accounts only for the memory actually consumed. That is, if a process allocates many pages, but uses only one page, only that page will be faulted-in increasing the writable memory count. We assume that processes do not read from allocated memory before writing to it first.

- Any shared memory that now becomes private. That is, when a page is mapped COW in a process, writes to that page produce a private, writable copy of the page in the process' address space. Writable memory precisely indicates how many COW pages became private, writable pages, and hence how much extra overhead is now being caused.

Writable memory does not account for read-only memory. This includes the executable pages of the application and libraries, and their respective read-only data. We do not count this in the "overhead" because sharing libraries (like libc) across the

Component	Data (KB)	Anon (KB)
Vanilla master	26.88	21.12
Vanilla worker	58.88	189.12
Vanilla total	**85.76**	**210.24**
sthread worker	20	24
sthread priv key	12	24
sthread sess key	24	40
sthread SSL_accept	36	16
sthread phase2	24	28
sthread total	**116**	**132**
Tagged memory		108
COWed tagged memory regions		364

Table 6.3: sthread and process memory breakdown for Apache.

whole system is likely, so the application should not be penalized for using them.[2] Note, however, that read-only memory will also include COW pages that have not yet been written to, which are still read-only to trap writes, and trigger copying. So, the actual writable memory may increase if the process writes to COW memory after the time of measurement. The increase is bounded by the size of the data segment, so one can calculate the worst case. This phenomenon is more likely to occur with `fork`, which marks all pages as COW, rather than sthreads, so our measurements for sthreads may be more precise than those for `fork`.

6.2.2 Apache's memory usage

Table 6.3 shows the memory footprint of the Apache worker, according to our first metric, both for the vanilla and sthread-partitioned version, when serving a request. We attempt to capture the per-client cost. The data column represents globals, initially marked as COW, which start using up memory once they are written to, triggering copies of pages. The anon column consists of private anonymous mappings: *i.e.,* heap and stack. The average size of an sthread, computed as the sum of all sthreads' data and anonymous memory sizes, divided by the number of sthreads (5), is about 50KB. This is a rough indicator as to how much memory use to expect per sthread. There are two main things to note: with sthreads, the overall memory use for globals increases, though the memory use for the heap decreases. Because globals are shared, whenever an sthread attempts to write to them, it will get its

[2] We actually link statically as our userspace implementation currently only supports such binaries, so the (fixed) overhead of library code is unique to applications. We still conduct our analysis on dynamic memory though to highlight the variable costs.

own copy. If all five sthreads write to the same set of globals, they will each incur the cost of copying the relevant COW pages, and the total cost increases compared to Apache's single worker that incurs the cost of only one page copy. Note that globals are often used by libraries at initialization. Because we checkpoint globals when `main` is called, libraries are likely uninitialized. As we start doing useful work, globals change, causing copying of COW pages and hence memory overhead. Perhaps checkpointing at a later time, or doing some "warm-up" work prior to checkpoint may reduce the need to copy COW pages, making the memory consumption smaller.

With regards to the heap, each sthread has its own so there is no sharing. The original heap used by Apache is now spread across all sthreads—it does not necessarily need to increase in size because of partitioning. Heap memory consumption actually *decreases* across an Apache worker's sthreads as compared with that for a single vanilla Apache worker process, because in vanilla Apache, `fork` maps the entire heap COW for the child, and thus, all future writes to the heap cause page copies in the child. Note that even if the parent writes, this causes a page copy, even if the child never reads from that memory, so this behavior can result in wasted work and a memory use increase. After `fork`, the parent has no writable memory until it starts executing and attempts to write to a COW page. This is why we include writable memory in the master as part of the memory overhead for vanilla Apache. Thus, `fork` incurs a cost when copying COW pages in the heap, but sthreads do not. sthreads also start with a smaller heap because they are given access to minimal memory. sthreads pay a price for globals, though a reduced price for heap, and the two almost balance each other.

Continuing to calculate the per-client cost, the overall sthread memory consumption we have accounted so far is 248KB, and for vanilla Apache, 296KB. Note, however, that another reason why sthreads use less heap is that some of this memory has now shifted into tagged memory regions, which we still have not accounted for. Our sthread Apache implementation uses 108KB of tagged memory per client. The total sthread per-client cost is therefore 356KB, to put it another way, 20% more than that of vanilla Apache. This overhead is caused by our partitioning and is not fundamental to sthreads—there could be a partitioning of Apache that requires fewer tagged memory regions, making the memory use closer to that of vanilla Apache. Note that we partitioned Apache in the most straightforward manner possible to provide secrecy and integrity for sensitive data, without specific regard for minimizing memory consumption.

Unfortunately, though, our sthread implementation has additional memory overhead. First, we require a one page process control block in userspace for each sthread. Since we use five sthreads per client, our per-client cost increases by 20KB. Second,

we emulate COW support for tagged memory regions, in userspace, by using `memcpy`, explicitly copying pages and causing unnecessary unsharing at sthread creation time. A more efficient implementation would actually mark pages as COW, avoiding memory use for pages that are only read from. Our Apache implementation uses COW rather heavily: we use 364KB of COWed data per client. Thus, our total sthread cost is 740KB. This is about 2.5 times the cost of Apache. Note that our COW implementation can be significantly improved. For example, we have a 244KB tagged memory region, all of whose pages are COW, to which only 44KB are ever written. Thus, our overhead of copying COW pages explicitly with `memcpy` is 82% for that tag, because we could have shared, rather than copied, 200KB of data. In practice though, because we copy all data, we never reference the old buffer again, which can be swapped out to disk to increase RAM availability.

Note that we have not yet accounted for in-kernel memory use. The kernel will need a process control block for each sthread, though this is less than 1KB in size. Furthermore, it will need a page table, though this grows more slowly than userspace memory. We have shown that much of the memory use comes from allocating large tagged memory regions, so we expect the kernel overhead to be negligible in such cases.

Figure 6.11 shows the total (userspace and kernel) physical memory consumption, as per our second metric, of vanilla Apache and the sthread-partitioned Apache as the number of concurrently served clients increases. The overhead is a factor of 2.2 on average. Note that this is slightly less than our 2.5 estimate, based on our first metric, for the following reasons. First, we are measuring memory use in a different way, using `free`, a command which reports the total used memory, subtracting any memory used for caching (*e.g.,* disk buffers). Second, we already noted that we underestimated, in our first metric, the memory use of `fork` because there could be more copying of COW pages (and hence, increase in the number of allocated pages) after measurement. Third, when sthreads are in a recycled state, their heaps are unmapped, making their memory consumption smaller.

For sthread-partitioned Apache, less than 800KB are required per client, so with 1GB of RAM, one can serve over 1,000 concurrent clients. Note that if memory overhead is added, by, for example, a server-side web application, the sthread overhead will remain the same. That is, only the single post-handshake per-client sthread will grow, just as Apache's single per-client worker will grow. Our overhead is merely due to the splitting of the SSL handshake and does not affect the memory requirements of the HTTP client-handling code. Only if we choose to partition the server-side web application will memory usage grow.

We now summarize our findings for sthread memory overhead:

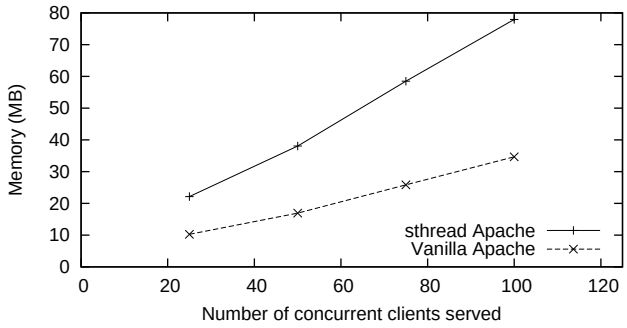


Figure 6.11: Memory usage as more clients are served concurrently.

- sthreads consume an average of 50KB of memory. They are most penalized from having to share globals COW. Their heap usage remains small because they start with a small private heap, and there is no copying of COW pages for the heap. These two effects balance out making the size of an sthread comparable to that of a process.

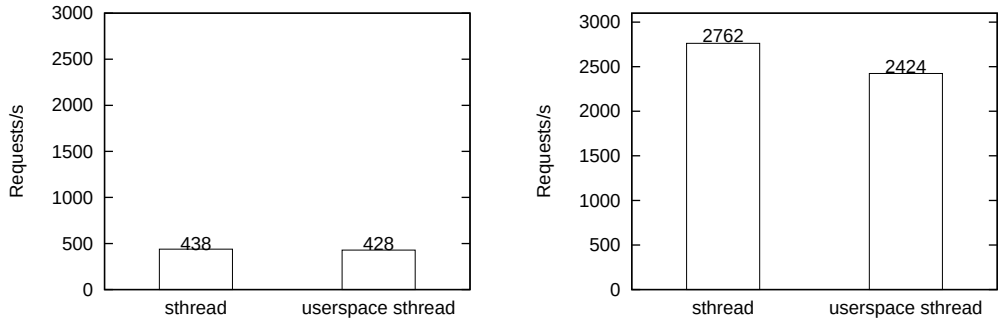
- Much of the sthread overhead comes from allocating large tagged memory regions and using COW. Thus, the application programmer has control over this overhead and can use sthreads in a way so as to decrease memory requirements, if he so desires. We already showed that writable memory affects performance too because it dictates how much scrubbing needs to occur. Therefore, programmers wishing to optimize sthread applications for speed, will also inherently improve the memory footprint too.

- Dormant sthreads in the recycled state are small because their heap is unmapped. Dormant processes will typically still have their heap mapped because they call `free` rather than `munmap`. Thus, a pool of sthreads can potentially use less physical memory than a pool of processes.

6.3 Newly written SSL web server

We now discuss the performance of an SSL web server written from scratch, contrasting some of its design features that affect performance with those of Apache. Our sthread-partitioned Apache implementation featured two main artifacts that greatly hindered its performance:

1. Sthreads were given access to many pages of writable memory (about 60). To simplify our implementation, we did not attempt to reduce memory usage. Hence, we ended up sharing many globals, which require a `memcpy` on recycle,



(a) No sessions cached.

(b) All sessions cached.

Figure 6.12: Performance of a newly written SSL web server using sthreads.

and allocating large buffers for arguments / return values, requiring a `memset` zero on recycle. We expect our newly written web server to use much less memory, as we can tightly define and control data structures.

2. Excessive levels (and numbers) of sthreads were used. We followed Apache's architecture of having multiple worker processes, and then created another level of child sthreads under each worker to handle requests. Because of the nature of the userspace implementation, it performs best when a master directly creates its child sthreads, rather than having a child sthread create an sthread, since that requires extra IPC to the master. Since in our newly written web server the master directly creates the sthreads that handle connections, we expect userspace performance to improve relative to the kernel version of sthreads.

We run the same throughput benchmark as we did with Apache, serving the same file on the same LAN, with and without SSL session caching. We are CPU-limited, and the sthread pool size is set to 50. Figure 6.12 shows the performance of our newly written web server running sthreads using the kernel and userspace implementation. The latter is only 2–12% slower than our kernel variant, depending on whether SSL sessions are cached or not. This is quite an improvement compared to the slowdown of up to 29% experienced with Apache. We conclude that the userspace implementation may be an efficient solution when the application is rather "heavy-weight" (*e.g.,* does cryptography) and if it is designed following certain criteria (*e.g.,* no grandchild sthreads).

We now account for how much memory the kernel must scrub. Our web server uses only two sthreads per client, and for each client request, the kernel must restore a total of 16 pages worth of globals, and zero 8 pages. We create two tagged regions

Component	Data (KB)	Anon (KB)
Phase1	24	16
Phase2	40	20
Total	**64**	**36**
Tagged memory		8

Table 6.4: Hand written httpd memory use breakdown.

Experiment	Vanilla	sthread
ssh login delay (s)	0.145	0.146
10MB scp delay (s)	0.486	0.482

Table 6.5: OpenSSH performance.

of memory per client, so require zeroing two more pages. Thus, we need to `memset` zero 10 pages, and `memcpy` 16 pages.

Memory consumption Table 6.4 shows the detailed memory consumption of each sthread as measured by exmap (our first metric). sthreads again consume about 50KB of memory, mostly due to the overhead of writes to COW-mapped globals, and in contrast with Apache, we now have minimal overhead due to tagged memory because we carefully defined and controlled data structures. Our total per-client cost is about 112KB, not accounting for any per-client state in the master and kernel. Measuring total memory use with `free` (our second metric), our newly written web server needs a total of 124KB for each concurrent client we serve.

6.4 OpenSSH

We now assess whether our sthread-partitioned OpenSSH significantly delays the daemon's operation. We expect the sthread-incurred delay to be negligible given that most of OpenSSH's delay is due to public key operations and executing a shell. We perform two experiments. First, we measure the end-to-end latency a user experiences to log into a shell. Second, we perform a file transfer with scp. For the login case, we measure the time it takes to start the ssh client, connect to the server, login, and logout immediately by using `exit` as a shell. This includes any network latency, though our tests were performed on a LAN where hosts were directly connected via a switch in order to make delays minimal. The scp case is the same as the login case, though work is added to the session, namely, a file transfer. We ran on a 1Gbit/s LAN to make file transfer time minimal.

Experiment	Original	sthread	diff
100 large PNGs (1000x900) load delay (ms)	12,085	12,994	+8%
100 small PNGs (10x10) load delay (ms)	168	176	+5%

Table 6.6: Firefox performance.

Table 6.5 shows the delay for a login, and a 10MB scp file transfer, both for the original version of OpenSSH and our sthread-partitioned version. As expected, the overhead is negligible for OpenSSH's interactive use. The overhead in the scp case is negligible as the price of sthreads is paid only during authentication—after that, a connection is handled in one sthread, just like with vanilla OpenSSH. Similar results were obtained with privilege separation [57].

6.5 Firefox

We now examine the cost of sthreads on client applications. We applied sthreads to Firefox to isolate vulnerabilities in libPNG. That is, libPNG is only allowed to write to display buffers specifically created for images, and libPNG cannot access any other memory of Firefox. Our results regarding Firefox reinforce that the cost of sthreads is low in applications that perform complex computation, and client-side applications with GUIs often fall into this category. PNG image decompression is CPU intensive, so we expect that adding sthread support to it will not significantly slow down the overall browser. It will, however, add to its security. Table 6.6 shows how long it takes to load a page with no other content apart from 100 PNG images. We use two different images. One is a small all-black PNG we created for the purpose of this benchmark, and the other is a large image [29] taken from the Internet. We measured page display latency using the load time analyzer add-on for Firefox [22]. This measures the time from when the browser requests the page until it is fully rendered on the screen. Note that we run these tests by reading an HTML file on the local filesystem, so there is no network latency involved. Like the original, our implementation decompresses the PNG row by row incrementally, so that an image being downloaded from the network can be displayed as it is retrieved. An sthread is created for every PNG image and lives until the image is decoded. Each time PNG data arrives, the master must feed it into the decoder sthread, incurring a context switch. As a row is decoded, the sthread must signal the master to display the new data, causing another context switch. This means that we need to context-switch at least twice for each row, and for this reason, large PNGs represent an inefficient case for our implementation, stressing our context-switch overhead. This partly

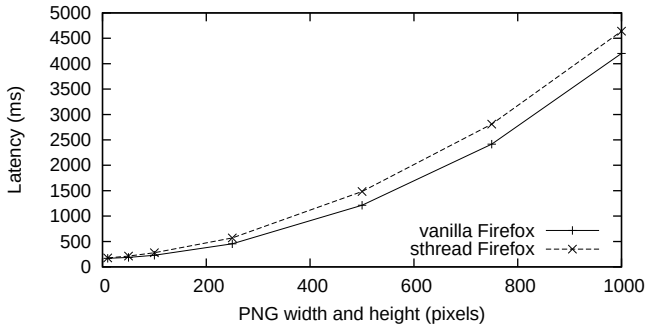


Figure 6.13: Latency of displaying PNG images of different size in Firefox.

explains why our performance decreases for PNGs with many rows (large height): a 8% overhead rather than 5%. The other reason is that we are able to decode small PNGs serially, so there is only one sthread running at any particular time, causing even fewer context switches. This is possible only for small images that can be fully decoded before Firefox starts processing the next one.

Figure 6.13 shows the latency required to display images of varying size. The experimental setup is as before, though we use an all-black square image of varying sizes. For small images, the overhead is lowest, because there are fewer rows to decode, and fewer sthreads involved, since images are decoded serially. This minimizes context-switch overhead. As the image size grows, more sthreads are required to decode the images; each takes more time to decode, so Firefox starts decoding them in parallel. The number of rows also increases, further increasing context-switch time and hence sthread overhead. The overhead, however, does not increase indefinitely as the image size grows. Indeed it begins to decrease, because more work is needed to handle and decode the larger images, and this work becomes the dominant cost. The peak overhead is with an image size of 250x250, where it reaches 26%. It then decreases to 10% as the image grows to 1000 pixels. Note that long and narrow images could perform differently as there is less (PNG) work per row, though the cost of displaying them still remains high, since it is proportional to the area.

If performance were the main requirement, perhaps one could decode the whole PNG in one sthread invocation to avoid context switching, at the cost of having to wait for the download to complete before displaying the image. A compromise would be to batch rows and decode (say) two at a time. This would improve performance and the user would still see the PNG loading incrementally, perhaps without even noticing the difference between increments of one row *vs.* two (or more).

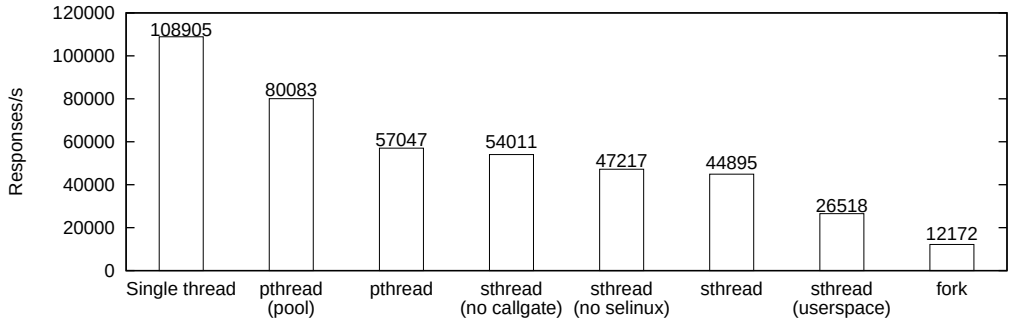


Figure 6.14: DNS performance.

Memory consumption According to our analysis with our first memory metric on Firefox, sthread memory overhead in Firefox is relatively low. Firefox consumes a lot of memory: about 26MB in our case. Because sthreads are only given access to the minimum set of memory they need, they are far smaller: to decode the large images, each sthread requires 88KB of memory, a mere 0.3% of the memory used by Firefox. For small images, there will only be one worker, as Firefox decodes them serially. For large images, we need multiple workers, so in our benchmark of 100 large images, our overhead increases to 30%. Note that these sthreads can be killed to reclaim memory once the page has finished loading. If memory is not low, they can be kept alive to speed up future invocations. With sthreads, the programmer can choose whether to trade memory for CPU speed and vice-versa.

6.6 DNS

We report the performance of our DNS server to highlight the cost of sthreads. A DNS server is typically written as a single-threaded process, which performs only a few operations per request, and is therefore very efficient. Our implementation instead must create an sthread for each request and hence greatly exposes the cost of sthreads, especially sthread recycling time.

Our DNS benchmark measures the throughput at which we can resolve hostnames. We have a workload of 100 parallel long-lived clients (requesting from a single host, as with Apache) querying for the DNS 'A' record of `www.darkircop.org`. On the server, we do not cap the number of clients served concurrently. We note, however, that the work per sthread is minimal, and the sthread often completes before the server starts handling the next request. Hence, the total number of sthreads barely grew, and indeed settled naturally at two during our tests. The tests were performed on a LAN and we were CPU-bound in all cases.

Figure 6.14 shows the performance of our DNS server when using different APIs. We compare against the same implementation using one thread, standard pthreads, and `fork`. In the case of one thread, we handle each request to completion before reading the next request. In the case of pthreads, we run two experiments: one with a thread pool which reuses threads, and another one where threads are created and destroyed each time. Sthreads outperform a `fork`-based implementation thanks to the recycling mechanism, which is cheaper than creating a new process (and taking page faults) each time. The userspace implementation suffers a 41% slowdown compared to the kernel version because the DNS server does very little work per request, exposing the cost of recycling sthreads in userspace using our unorthodox ways using `ptrace`, `shmctl` and `mmap`. Our userspace implementation therefore performs better for more complex servers (*e.g.,* an SSL web server).

We ran the benchmark with and without the use of the only callgate in our partitioning (resolve), to see how much it costs. With the callgate, users are isolated from each other and zone data is kept secret. Without the callgate, the latter property is not met, though performance increases by 14%, highlighting a trade-off between security and performance. The callgate requires extra context switching and sharing of memory buffers for arguments and return values, the latter requiring more memory scrubbing.

We compared our standard sthread DNS version running with full SELinux protection with one that merely uses `chroot` and `setuid` to drop privileges. SELinux adds 5% overhead in this case. This is the highest SELinux overhead we measured across all our applications. Note that `chroot` may be well suited for high performance sthreads that need not access to the filesystem at all: no files need to be copied in the chroot jail, thus avoiding configuration overhead, and no SELinux performance overhead is paid. Of course SELinux provides more protection, like IPC, though cheaper mechanisms like `jail`, firewalls, or grsecurity [23] can be used instead for equivalent security, especially when performance is most important.

The cost of the sthread itself (ignoring SELinux) in a lightweight application such as DNS is running 5% slower than using a freshly created pthread. Note that in this benchmark, sthreads spend about 6% of their time resolving TLB misses, according to the PAGE_WALK performance counter. This accounts for some of the difference between sthreads and pthreads. But if we reuse pthreads from a pool rather than creating them new each time, as we do with sthreads, the relative overhead of sthreads increases to 33%. This cost is due to the memory scrubbing required by sthread recycling.

The cost of sthreads is evident when comparing to the single-threaded version, where the sthread-partitioned version is 2.4 times slower. Note that of this slowdown,

Component	Data (KB)	Anon (KB)
single thread	13.88	30.12
pthread pool	13.88	42.12
fork master	8	12
fork callgate	9.88	10.12
fork worker	9.88	18.12
Total	**27.76**	**40.24**
sthread master	20	24
callgate	12	12
sthread worker	16	12
Total	**48**	**48**
Tagged memory		8

Table 6.7: Memory cost breakdown of DNS server when using different APIs.

almost half (45%) is due to handling users concurrently (*e.g.,* using a thread pool) rather than handling one at a time in a straight line of code, as the single-threaded version does. Thus, not all of the cost comes from the sthread mechanism per se. Of the sthread-specific cost, 19% is accounted for by the callgate, 7% is consumed by SELinux, 11% from context switching, and the rest is due to memory scrubbing. Thus, for non-threaded applications, the slowdown of sthreads comes partly from splitting the code into multiple threads, and partly from sthread recycling, *i.e.*, the added isolation.

Memory consumption Table 6.7 shows the memory consumption, as per our first metric, of the several implementations of the DNS server. With a single thread, the total consumption is 44KB, and with a thread pool, it is 56KB. With sthreads, the memory consumption inflates to 152KB (we use two workers). We are therefore about 3.5 times more expensive than a single thread, and about 2.7 times more costly than a thread pool. The per-client cost of our DNS server is rather low, about 40KB, though we only use two clients, so our overall cost is amplified due to the relatively high fixed costs of our master and callgate.

A similar implementation using `fork` incurs less memory overhead than sthreads, costing a total 96KB for two workers, making sthreads 1.6 times more expensive. sthreads pay a high price for copying COW-mapped globals but a low price for the heap. Because the heap is small in our DNS example, the overhead of `fork`'s copying of COW-mapped pages does not outweigh the cost we pay for globals with sthreads. Actually, for the worker sthread it does, but our overall cost is hindered by the master which uses extra heap for sthread and tag bookkeeping information. If the

DNS server had many parallel clients, the fixed cost of the master and callgate would be mitigated, making the relative memory overhead lower. sthreads further waste memory through internal fragmentation: tagged memory regions must be a multiple of the page size in length. Even though the arguments to our sthread are only a couple of bytes, we still need to allocate a whole page. Thus, for applications with tiny heaps, our sthread implementation becomes the main source of overhead—not the programmer's partitioning and use of tags. To improve, we need to lower the "minimum size" of an sthread. We can do this by avoiding the copying of COW-mapped globals due to library initialization by checkpointing later. Furthermore, we can trim the stack at checkpoint time—note that our anon mappings are often 12KB: this is because we checkpoint three stack pages so all sthreads inherit that, and occupy that space.

6.7 Fundamental limits and possible enhancements

We now discuss the fundamental costs of sthreads, regardless of their implementation. There are two main costs associated with sthreads: memory cleanup and context switch time.

Because we assume that all data written by an sthread is sensitive, when an sthread exits, we must scrub all that memory. One way to think about this activity is that we repeatedly unmap the process's heap on exit, and then allocate a new one when the process starts. Behind the scenes this means that we must zero deallocated memory so that it can be reused when allocated again without it disclosing any sensitive information. We are therefore limited by the speed at which we can zero memory. Unfortunately, zeroing a large number of pages is relatively slow. Perhaps we could have hardware support for quickly zeroing memory, or perhaps we can trick the DMA controller into doing so in the background, or even encrypt memory! Sometimes, as with COW mappings, rather than zeroing memory, we must reinitialize it to some known state. This operation, `memcpy`, is unfortunately even slower. Again, we probably need special hardware support to get any gains on this.

Another overhead of sthreads is context-switch time, although we found this to be short in practice. Because we use page-based memory protection, we must flush the TLB on each context switch. Where supported, one could use a tagged-TLB [39] to flush only those portions of the memory map that really change. This would mitigate the cost of a full TLB flush. For example, the `.text` segment, which does not change but is frequently accessed, will not suffer from TLB misses. Perhaps one could attempt other tricks such as using the "global page" bit on the page table to avoid a page from being flushed, or using `invlpg` to invalidate individual page

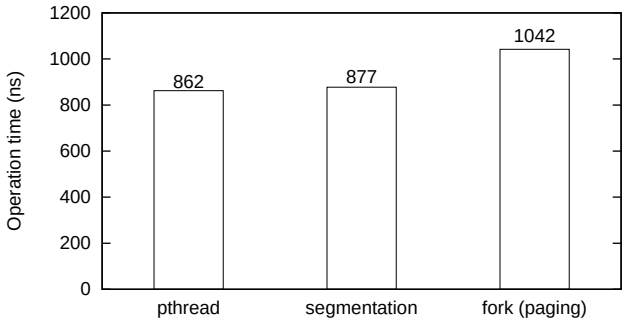


Figure 6.15: The context-switch overhead of sthreads using segmentation is only 2% more than that of pthreads.

entries.

An alternative to paging for memory protection is segmentation. Segmentation is largely unused today, so our natural inclination was to implement sthreads using paging, in order to change less of the OS. With segmentation, it is possible to context-switch without flushing the TLB. This technique would make the context-switch overhead of sthreads more similar to that of pthreads, rather than processes. We would then partition the virtual memory space into multiple segments, and each sthread's memory would be contained within a segment. Segments can be specified in the Local Descriptor Table (LDT), which contains information about the segment's base, limit, permissions and identifier. To context switch to a new sthread, the kernel would modify the LDT of the process, pointing it to the segment information for that sthread. This would effectively change the sthread's segment. The sthread would only be able to address its own memory (defined by the base and limit) and not that of other sthreads. No TLB flushes would be required since segmentation will limit the address space of the thread. Some TLB misses will occur because we are increasing the virtual memory usage, and we are now accessing more virtual addresses. Figure 6.15 shows the context-switch overhead of processes, a toy sthreads implementation using segmentation, and pthreads. Context-switching via segmentation is remarkably fast, though still slower than pthreads due to the extra TLB (capacity) misses. The main drawback, though, is that we would limit the virtual address space each sthread has access to, and thus limit the number of sthreads we can run per process. Given the relatively small performance gain and the quite significant design change, we did not go down this route.

Nevertheless, an interesting application remains for segmentation: callgates, using x86 callgates [25]. An x86 callgate defines an entry point that runs in a more privileged ring. The operating system runs in ring zero, and userspace in ring three, leaving other rings unused. We could set sthreads to run in ring three and callgates

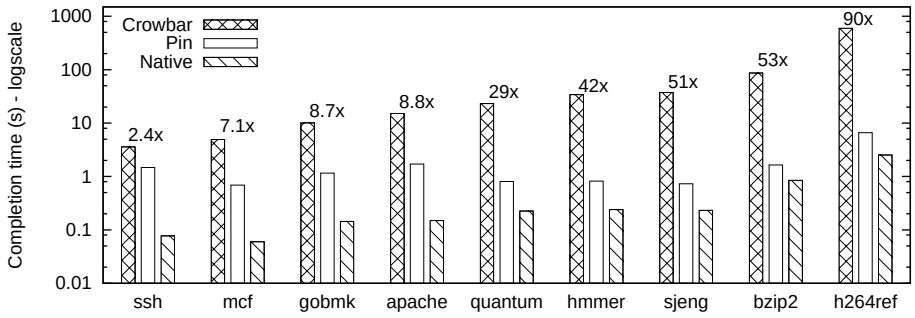


Figure 6.16: Execution time of Crowbar. The number over the bars is the ratio between Pin and Crowbar. The y axis is in log scale.

in ring two, and use an x86 gate to jump between sthreads and callgates. The callgate's memory could live in segments marked as ring two so that sthreads could not access them. When a callgate would start execution, though, it would be able to access those segments (and hence memory), since it would run in ring two. With a simple and cheap x86 `call` instruction, we could then enter a callgate entirely in userspace, and we would automatically receive the memory privileges we want. This approach is obviously much faster than having to enter the kernel and context switch using paging. In fact, we can do it over thirty times faster in a toy benchmark that enters a callgate and returns immediately. The drawback of this particular scheme, though, is that callgates will all share the same memory privileges, because they will all run in the same ring (two). If we in future implement "sthreads-lite", where there is a single lump of trusted code per application, and the rest is sthreads, then x86 gates seem a promising performance enhancement.

6.8 Crowbar

The Crowbar tools are used by developers to add sthread support to existing applications. We now discuss the delay developers may experience when tracing such applications. Because Crowbar tools only impact development time, their performance merely needs to suffice not to slow developers down. If a trace with Crowbar can be obtained in several seconds, performance is adequate, since a programmer would typically need to invest several hours in reading the code to obtain the information derived by Crowbar. One would expect that Crowbar slows the applications it instruments, since it instruments all memory operations.

Figure 6.16 shows the execution time of Crowbar when run on a number of applications in the SPEC benchmarks [66]. Note that we ran these tools with our own inputs, and not those provided by SPEC, since we required short completion

consisting of a single run rather than many complex iterations as benchmarks typically do. We also include Apache and OpenSSH, to show the overhead that would be experienced when dealing with a network server. For each application, we plot three bars: one indicating the total completion time of Crowbar as experienced by developers, one the time required by Pin with no instrumentation (which shows a baseline), and one the uninstrumented execution time. Pin instruments each basic block it encounters once and then caches it, reusing the cached instrumented version on any subsequent runs. In other words, Pin's instrumentation overhead is visible only the first time a basic block is run, and subsequent runs are cached instrumented code. This instrumentation overhead is depicted by the Pin bar, and any users of Pin cannot do better than this; hence this is our baseline.

Crowbar's overhead varies depending on the application. From the graph, two main classes of applications are seen. On the left we see "less iterative" applications. That is, applications that run through a straight line of code and exit. In these, the bulk of the cost is from Pin's instrumentation phase, since many new basic blocks are encountered, which must be instrumented, so the instrumented code cache is largely unused. The right side of the plot hosts "more iterative" applications. These revisit many basic blocks multiple times, and hence Pin's instrumentation overhead is amortized since it uses its cache the second (and later) times round. Our Crowbar implementation, however, does its job on every iteration and its relative overhead therefore appears greater. We note that when obtaining a trace for OpenSSH and Apache/OpenSSL, it was necessary to run through the code in a straight line rather than repeat an operation multiple times. This is so because one run will reveal all the memory locations that code needs—multiple runs would reveal the same information. This explains why Crowbar is particularly slow with these benchmarks which tend to be iterative (which is natural for benchmarks).

Note that the absolute time for obtaining a trace is acceptable in all cases we measured, and this is what matters, because this is how long developers must wait. The maximum wait time for the developer was ten minutes, and on average, one would need to wait 90 seconds. Despite the large slowdown factor of Crowbar, the absolute time remains low because the uninstrumented execution time is tiny, so in essence we are not too concerned if Crowbar is 90 times slower than uninstrumented! We have not run Crowbar on interactive applications (our Firefox/libPNG partitioning was manual, as memory dependencies were trivial) and expect these applications to be most problematic, since developers will need a lot of patience as applications respond slowly. We expect the experience to be similar to running an X11 application over a WAN.

6.9 Summary

Compared to threads, sthreads have a large creation overhead, though the context-switch overhead is not so great. To improve creation, we keep a pool of long-lived sthreads, and recycle them on reuse, bringing them back to their pristine state. This significantly improves performance, making sthread recycling competitive when compared to thread pools, and indeed faster than creating new threads, let alone new processes. The cost of creating sthreads is now reduced to zeroing any used memory (privacy), restoring original memory values in case of COW (integrity), and overhead due to TLB misses after context switching. This latter cost might be eliminated by using a segmentation-based implementation, though we expect only a marginal gain.

In complex applications such as SSL Apache, the cost of sthreads is relatively little, ranging from 6% to 25%. The cost of sthreads becomes a bottleneck only when applications perform very little work, such as in a single-threaded DNS server. Low memory consumption was not a focus in partitioning legacy code, so we never tried to improve on this aspect. As is, in some cases, applications suffer from a 2.2 times memory increase, though much of this can be optimized by the application programmer, by, for example, using memory tags more carefully. Nevertheless, the absolute amount of memory consumed remains low in all cases, leaving the CPU as the limiting resource for scalability and performance. Our userspace implementation is suitable for complex applications, where slowdowns compared to our kernel version are as low as 10%, though we have measured slowdowns of up to 41% in cheap applications.

Finally, we show that Crowbar performs well enough not to slow programmers' development. Indeed, in very little time it gathers information that would otherwise be time-consuming to obtain by manual code study.

Chapter 7

Conclusion

In this thesis, we have tackled the problem of how to allow programmers to apply the principle of least privilege in applications, with least code changes, focusing largely on existing code on today's commodity OSes. We presented sthreads, a set of abstractions that programmers can use to secure their applications. Sthreads allow a programmer to create default-deny compartments which give the programmer explicit control over what memory, file descriptors and system calls are available to each compartment. By limiting the access of each compartment, the programmer denies attackers benefit from exploiting them. When sthreads are applied correctly and thoroughly, the programmer can narrow the attack surface from the whole application to only privileged compartments, which we refer to as callgates. We achieve all these goals without significantly hindering performance, for the range of applications we have considered.

To help apply sthreads to existing legacy code, we provide Crowbar, a tool that tells the programmer which privileges each compartment needs. The programmer therefore only needs to identify compartments in an application and then use Crowbar to perform the tedious work of identifying the memory privileges for each compartment. The programmer can then grant these. Crowbar therefore makes it practical for programmers to partition existing code with sthreads.

7.1 Contributions

We show that simple OS extensions can allow programmers to greatly improve isolation in today's applications, by making applications adhere more closely to the principle of least privilege. Our sthread mechanisms incur an acceptable performance cost: for example, sthread-partitioned Apache pays only a 6–25% performance overhead, while being able to protect sensitive data from a man-in-the-middle that can

also exploit most of the web server. If necessary, sthreads can be implemented fully in userspace at some performance cost, showing that it is possible to improve application isolation significantly, even without requiring new kernel functionality (on Linux). Our APIs closely resemble existing OS concepts, hopefully making them simple for programmers to understand and use.

We provide systematic aids for partitioning existing applications. We note that the difficulty in partitioning applications for enhanced isolation is in determining the privileges each compartment requires, and its data dependencies on other compartments. Our Crowbar tool provides the programmer with this information. For example, when partitioning OpenSSH, Crowbar pointed out only 6 source files out of OpenSSH's total codebase of 120 files, significantly narrowing down the amount of code we had to study. Furthermore, it pointed out exact source code lines and memory objects that we should concentrate on. We therefore provide practical means for compartmentalizing existing code.

We show that little changes are necessary to existing code in order to partition existing applications, and most of these are guided by tools. In our experience, we had to change at most 1,700 lines of application code (for Apache).

sthreads alone, without Crowbar's help, are remarkably simple to use on some library code, because memory dependencies between the library and application are kept to a minimum and well specified. Our experience with Firefox and libPNG suggests that partitioning existing code at a library boundary is a very well suited task for sthreads, thanks to the well defined boundaries, APIs, and memory dependencies imposed by some libraries. Indeed, we only had to change 284 lines in Firefox's huge codebase, and were able to do so trivially, without aid from Crowbar. Applying sthreads to libraries has practical benefits, first, because vulnerabilities do occur in them, and second, because users of libraries seldom audit library code, which may not even be available. Thus, we believe that sthreads are a major contribution for isolating library code.

sthreads highlight key missing features in UNIX's APIs necessary for applying least privilege. `fork`, could, in principle, be used for process-level isolation, though it lacks two important properties for being practical. First, it lacks privacy support because a child can read the parent's memory as at the time of `fork`. `fork` does provide integrity support thanks to COW, since a child cannot write to its parent memory, though sthreads importantly add privacy support too. Second, `fork` is not fast enough for creating short-lived processes. sthreads cure this by using the `checkpoint` and `restore` mechanism we presented. Indeed, the viability of process-like isolation has been doubted in the past due to the performance and memory overhead of processes [32, 18]. With sthreads, we show that we can reduce CPU

overhead enough to make process-like isolation practical, and show that we have not experienced memory exhaustion with the real applications we tested.

7.2 Reflections

We started off by implementing the sthread API fully in the kernel, but ended up with a relatively efficient implementation fully in userspace. So what is the ideal kernel API for allowing programmers to adhere to least-privilege? We started our journey thinking that the basics were sthreads, tagged memory, and callgates. We therefore implemented these primitives in the kernel, built the rest in userspace, secured real applications, and demonstrated that it all works in practice. We quickly realized that performance was a problem and discovered reused callgates, which also made us realize that you can implement a callgate as an sthread, namely by creating a long-lived sthread and using locking to invoke it. Thus, the only low-level primitives necessary became sthreads and tagged memory, *i.e.,* compartments and memory protection. We then realized that more performance can be gained if sthreads are recycled, by cleaning up the memory map upon exit and recycling the sthread. We knew that `fork` was too slow for creating compartments, so we always stayed away from a userspace implementation, but we then asked ourselves: what if we can recycle processes like we recycle sthreads? Perhaps that would make it possible (performance-wise) to have an implementation completely in userspace. And so it was. The answer to our API question therefore might be: we need nothing new from the kernel, beyond what Linux offers today.

Not quite. Our userspace implementation is built upon layers of primitives that ultimately provide sthreads, tagged memory and callgates. Tagged memory is built using `mmap` and System V shared memory, both of which are standard UNIX APIs. Callgates are built using sthreads. Sthreads are built from a rather peculiar primitive, nowhere near our original APIs. This primitive essentially is "checkpoint and restore". We use `fork` to create compartments, but the essence of the problem is how to recycle them, making compartment "creation" fast enough for practical use. Effectively we checkpoint the application at the very beginning, and to create a compartment, we restore. We can provide this primitive fully in userspace, but only thanks to `ptrace` and System V shared memory trickery—it is not an elegant solution. To build a cleaner system, this checkpoint and restore abstraction is the only one we would need from the kernel. Once the kernel provides it, one can implement sthreads by checkpointing at `main` and restoring to create a new sthread. In fact, in our kernel implementation, the kernel does not even know what an sthread is. With these basic building blocks of checkpoint and restore, other security systems

could be implemented, too. For example, one can implement a system that relaxes default-deny by checkpointing just *after* the application loads its configuration, but *before* it talks to the network. This way, all child sthreads receive the configuration memory by default without requiring tagging—this approach could be a big win when partitioning legacy code.

Interestingly, when looking back, from the operating-system perspective, our work evolved into something even more primitive than what we started with. We thought that sthreads, tagged memory, and callgates were fundamental abstractions. We were wrong. The real primitive we were looking for was something as simple as checkpoint and restore. This concept differs greatly from the abstractions we initially proposed. Its motivation, too, is entirely different from our original one—performance first, rather than security. Only now do we realize that we actually have been building this system top-down when we had intended to build it bottom-up.

7.3 Future directions

We now discuss some of the questions that arose in this work, some of which address its limitations.

Can we make callgates harder to exploit, rather than relying solely on manual auditing? Perhaps we can apply exploit prevention techniques such as CFI [4] to callgates. These techniques typically incur a run-time overhead, but we will pay it only for a narrow part of the application (callgates). Perhaps we could rewrite our callgates using a safe language (*e.g.,* Java). This way we could (most likely) prevent exploits from occurring. We might be able to afford doing so because callgates are typically small, so we only would need to rewrite a relatively small portion of the application.

Can we ensure that the information obtained from Crowbar is exhaustive? To improve Crowbar we can explore whether we can use a hybrid of static analysis and run-time instrumentation to determine permissions that are just right rather than too restrictive or permissive.

Can we use tools to automatically transform code based on some partitioning diagram? Perhaps we can adopt model-driven architecture techniques by supplying Crowbar with a security diagram which will be used to transform existing code to a matching implementation. Rather than manually partitioning, one merely needs to worry about the design.

How do we partition applications securely? We solved the problem of not having to worry about writing a bug-free *implementation*, though we still require the *design* (*i.e.,* partitioning) to be bug-free. Are there rules for partitioning? Are there

"equivalence classes" that tell us when we can combine or split sthreads? Are there design rules to avoid problems like oracles and man-in-the-middle attacks, as shown in the SSL Apache case? Can we prove that a partitioning is secure with respect to some properties?

What tools could assist us when partitioning legacy applications? We have shown that Crowbar greatly helps for partitioning existing code. Our experience also seems to suggest that even with simple primitives like sthreads, partitioning existing code still remains difficult. Hence, future research in tools, rather than APIs, might be the right path for solving the problem of partitioning existing applications. It is also remarkable how useful Crowbar was, despite it being a relatively basic tool. This makes us think that there is great potential in trying to devise more powerful tools.

Is there a less secure variant of sthreads that can be trivially applied to existing code? For example, is there a drop-in replacement for `fork` that provides more security (`sfork`)? Consider a web server forks a child for each request but reuses children. Perhaps we can implement a fast version of `fork` so that children are always new, ensuring the server's privacy. An interesting observation is that parent processes typically have control data that must not be tampered with (*e.g.,* configuration) whereas children generate sensitive data (*e.g.,* user sessions). If this observation holds generally, `fork` may be a good enough solution since it protects parents (COW) and keeps children's data private (new process each time).

Can we exploit the processing capacity of multi-core machines for security? Many client machines are multi-core and the CPU utilization is seldom 100%. Maybe this spare capacity can be used to secure client applications without sacrificing performance. We noted that fundamental limits exist for ensuring privacy, such as scrubbing memory pages. Perhaps we can have spare CPUs constantly scrubbing memory. Similarly we can have them creating a pool of compartments that can be readily used when needed. Such techniques could minimize the delay imposed by security systems, virtually eliminating their cost.

Can hardware support enhance the performance of sthreads? We noted that much of the cost of recycling an sthread comes from zeroing memory. Indeed, zeroing sensitive memory is likely an operation that is needed in any security system that enforces privacy. Being a (logically) simple operation, it may be possible to build fast hardware support for zeroing large portions of memory. This will definitely boost the performance of sthreads. Hardware COW support would be great too!

Can we use concepts (and syntax) from object-oriented languages to specify sthread policies? For example, we can use the `private` keyword of C++ to denote memory not visible to other sthreads. Similarly, the `public` keyword can be used to expose or share data. C++ enforces encapsulation at compile-time, and with

sthreads, we can provide that protection at run-time too, as is done, for example, in Java. Thus, programs that already encapsulate sensitive data will benefit from run-time protection, without requiring any code changes.

Can we apply the concepts of sthreads to languages other than C? For example, we could implement an sthread-like system for PHP, to provide isolation in web applications. Rather than giving PHP scripts access to all variables defined in an application, we can protect certain ones from being accessed from certain contexts. This will limit harm caused by PHP-code-injection vulnerabilities.

7.4 Lessons

To meet high security goals, it is not absolutely necessary to revolutionize applications and operating systems. Our sthread system can secure applications by using simple concepts that are not far off existing ones. This means that existing applications will often require only a few changes to migrate from their current use of (say) processes and threads, to sthreads. To provide security, operating systems merely need to extend their functionality, building on existing abstractions, without requiring any core changes. Indeed we show that even without kernel changes, we can still support sthreads on Linux.

Unfortunately, no matter how simple a security mechanism is, it still remains remarkably difficult to apply it to existing code. Translating an application to use a new security mechanism is more similar in difficulty to translating it into another language, rather than porting it to another operating system, though we originally thought that our effort would resemble the latter. With sthreads, we change the semantics of processes—now only *some* memory is available to code. We originally hoped we could get away with changing calls from `pthread_create` to `sthread_create`, as though porting to a different platform. Instead, in practice, it was more like translating applications from C to Java—we had now introduced the notion of private and public members, with access restrictions enforced at runtime. We approached this difficulty with new tools rather than different primitives. We conclude that simplicity is truly paramount for the design of practical secure programming primitives. Furthermore, tools, rather than primitives, seem to be the answer to the problem of how to partition existing code; and the latter appears to be the *real* problem.

Bibliography

[1] Apache-SSL buffer overflow vulnerability. http://www.securityfocus.com/bid/4189.

[2] COVTOOL—Free test coverage analyzer for C++. http://covtool.sourceforge.net/.

[3] OProfile - A System Profiler for Linux. http://oprofile.sourceforge.net/.

[4] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *CCS*, 2005.

[5] AIST. Openssl insecure protocol negotiation weakness. http://www.securityfocus.com/bid/15071.

[6] Z. Aral, J. Bloom, T. Doeppner, I. Gertner, A. Langerman, and G. Schaffer. Variable weight processes with flexible shared resources. In *USENIX*, 1989.

[7] E. Benson. OpenSSH Remote Root Authentication Timing Side-Channel Weakness. http://www.securityfocus.com/bid/7482.

[8] J. Berthels. Exmap. http://www.berthels.co.uk/exmap/.

[9] A. C. Bomberger, W. S. Frantz, A. C. Hardy, N. Hardy, C. R. Landau, and J. S. Shapiro. The keykos nanokernel architecture. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 95–112, Berkeley, CA, USA, 1992. USENIX Association.

[10] N. Breese. VMware vmware-config.pl SSL Key File Permission Weakness. http://osvdb.org/show/osvdb/27418.

[11] Brice Canvel, Alain Hiltgen, Serge Vaudenay, and Martin Vuagnoux. OpenSSL CBC Error Information Leakage Weakness. http://www.securityfocus.com/bid/6884.

[12] P. Broadwell, M. Harren, and N. Sastry. Scrash: a system for generating secure crash information. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 19–19, Berkeley, CA, USA, 2003. USENIX Association.

[13] D. Brumley and D. Boneh. Openssl timing attack rsa private key information disclosure vulnerability. `http://www.securityfocus.com/bid/7101`.

[14] D. Brumley and D. Song. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security*, 2004.

[15] J. Chase. *An Operating System Structure for Wide-Address Architectures*. PhD thesis, University of Washington, 1995.

[16] W. Cohen. Kprobes and the Linux kernel, 2005. `http://www.redhat.com/magazine/005mar05/features/kprobes/`.

[17] D.Bell and L.LaPadula. Secure computer systems: Unified exposition and multics interpretation. In *Technical Report ESD-TR-75-306, MTR-2997, MITRE, Bedford, Mass*, 1975.

[18] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the asbestos operating system. In *SOSP*, 2005.

[19] K. Elphinstone, G. Klein, P. Derrin, T. Roscoe, and G. Heiser. Towards a practical, verified kernel. In *HOTOS'07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, pages 1–6, Berkeley, CA, USA, 2007. USENIX Association.

[20] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. Necula. XFI: Software guards for system address spaces. In *OSDI*, 2006.

[21] C. Evans. LibPNG Graphics Library Multiple Remote Vulnerabilities. `http://www.securityfocus.com/bid/10857`.

[22] Google. Load Time Analyzer 1.5. `https://addons.mozilla.org/en-US/firefox/addon/3371`.

[23] grsecurity. grsecurity. `http://www.grsecurity.net/`.

[24] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical taint-based protection using demand emulation. In *EuroSys '06: Proceedings of the*

1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006, pages 29–41, New York, NY, USA, 2006. ACM.

[25] Intel. Intel architecture software developer's manual, 1999.

[26] Intel Corporation. Intel 64 and IA-32 Architectures Optimization Reference Manual, 2008.

[27] ISS. OpenSSH challenge-response buffer overflow vulnerabilities, 2002. `http://www.securityfocus.com/bid/5093`.

[28] M. Ivaldi. OpenSSH-Portable Existing Password Remote Information Disclosure Weakness. `http://www.securityfocus.com/bid/20418`.

[29] J. Jameson. Test image for libPNG benchmark. `http://www.playboy.com/`.

[30] D. Kilpatrick. Privman: A library for partitioning applications. In *USENIX Security, FREENIX Track*, 2003.

[31] S. Krahmer. OpenSSH Authentication Execution Path Timing Information Leakage Weakness. `http://www.securityfocus.com/bid/7343`.

[32] M. Krohn. Building secure high-performance web services with OKWS. In *USENIX*, Boston, MA, June 2004.

[33] M. Krohn, P. Efstathopoulos, C. Frey, F. Kaashoek, E. Kohler, D. Mazières, R. Morris, M. Osborne, S. VanDeBogart, and D. Ziegler. Make least privilege a right (not a privilege). In *HotOS*, 2005.

[34] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard os abstractions. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 321–334, New York, NY, USA, 2007. ACM.

[35] M. Kuhn. OpenSSH PAM conversation memory scrubbing weakness, 2003. `http://www.securityfocus.com/bid/9040`.

[36] lamagra. Format Bugs: What are they, Where did they come from, How to exploit them. `http://hackerproof.org/technotes/format/format_bugs.txt`.

[37] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO*, 2004.

[38] L. K. C. Leighton. dynamic context transitions. `http://lists.samba.org/archive/samba-technical/2004-November/037870.html`.

[39] M. Lewis. The difference between AMD and Intel processors. http://amd.vendors.slashdot.org/article.pl?sid=06/05/15/1750200.

[40] J. Leyden. Pentagon hacker Analyzer suspected of $10m cyberheist. Credit card scam exposed, 2009.

[41] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *USENIX*, 2001.

[42] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[43] J. McDonald. OpenSSL SSLv2 Malformed Client Key Remote Buffer Overflow Vulnerability. `http://www.securityfocus.com/bid/5363`.

[44] Microsoft Security Bulletin. Microsoft VM Bytecode Verifier Execute Arbitrary Code. `http://osvdb.org/show/osvdb/2969`.

[45] R. Morris. Berkeley Sendmail DEBUG Vulnerability. `http://www.securityfocus.com/bid/1`.

[46] R. Morris. BSD fingerd buffer overflow Vulnerability. `http://www.securityfocus.com/bid/2`.

[47] A. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM TOSEM*, 9(4):410–442, 2000.

[48] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2005)*, 2005.

[49] NISCC and Stephen Henson. OpenSSL ASN.1 Parsing Vulnerabilities. `http://www.securityfocus.com/bid/8732`.

[50] Nsfocus Security Team. Microsoft Windows 9x / Me Share Level Password Bypass Vulnerability. `http://www.securityfocus.com/bid/1780`.

[51] OpenBSD. W⊕X. `http://www.openbsd.org/33.html`.

[52] OSVDB. Open Source Vulnerability Database. `http://osvdb.org`.

[53] perfmon2. perfmon2: the hardware-based performance monitoring interface for Linux. `http://perfmon2.sourceforge.net/`.

[54] J. Pol. OpenSSH Channel Code Off-By-One Vulnerability. `http://www.securityfocus.com/bid/4241`.

[55] H. Pötzl. Linux-vserver technology. `http://linux-vserver.org/Linux-VServer-Paper`, 2004.

[56] N. Provos. Improving host security with system call policies, 2002.

[57] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *USENIX Security*, 2003.

[58] Rembrandt. OpenSSH S/Key remote information disclosure vulnerability, 2002. `http://www.securityfocus.com/bid/23601`.

[59] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[60] M. D. Schroeder and J. Saltzer. A hardware architecture for implementing protection rings. *CACM*, 15(3), March 1972.

[61] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307, New York, NY, USA, 2004. ACM.

[62] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 16–16, Berkeley, CA, USA, 2001. USENIX Association.

[63] J. S. Shapiro, J. M. Smith, and D. J. Farber. Eros: a fast capability system. In *In Symposium on Operating Systems Principles*, pages 170–185, 1999.

[64] Solar Designer. Getting around non-executable stack (and fix). `http://ebook.security-portal.cz/book/advanced_overflows/ret-libc.txt`.

[65] Solar Eclipse. Apache / OpenSSL remote exploit for CAN-2002-0656. `http://www.phreedom.org/solar/exploits/apache-openssl/`.

[66] SPEC. SPEC CINT2006 Benchmarks, 2006. `http://www.spec.org/cpu2006/CINT2006/`.

[67] P. Starzetz. Linux Kernel do_brk Function Boundary Condition Vulnerability. `http://www.securityfocus.com/bid/9138/`.

[68] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Trans. Comput. Syst.*, 23(1):77–110, 2005.

[69] A. Szombierski. Linux Kernel Privileged Process Hijacking Vulnerability. `http://www.securityfocus.com/bid/7112/`.

[70] P. Team. PaX address space layout randomization.

[71] The Bunker. OpenSSL SSLv3 Session ID Buffer Overflow Vulnerability. `http://www.securityfocus.com/bid/5362`.

[72] F. Tip. A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands, The Netherlands, 1994.

[73] Vlastimil Klima, Ondrej Pokorny, and Tomas Rosa. OpenSSL Bad Version Oracle Side Channel Attack Vulnerability. `http://www.securityfocus.com/bid/7148`.

[74] VMware. VMware and the National Security Agency team to build advanced secure computer systems, 2001. `http://www.vmware.com/pdf/TechTrendNotes.pdf`.

[75] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *SOSP*, 1993.

[76] S. Wallace and K. M. Hazelwood. Superpin: Parallelizing dynamic instrumentation for real-time performance. In *CGO*, pages 209–220, 2007.

[77] M. V. Wilkes and R. M. Needham. The Cambridge CAP Computer and Its Operating System. 1979.

[78] T. Wolff. Fetchmail fetchmailconf Race Condition Password Disclosure. `http://osvdb.org/show/osvdb/20267`.

[79] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. Hydra: the kernel of a multiprocessor operating system. *Commun. ACM*, 17(6):337–345, 1974.

[80] M. Zalewski. SSH CRC-32 compensation attack detector vulnerability, 2001. `http://www.securityfocus.com/bid/2347`.

[81] S. Zdancewic, L. Zheng, N. Nystrom, and A. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *SOSP*, 2001.

[82] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI*, 2007.