Wedge: Splitting Applications into Reduced-Privilege Compartments

Andrea Bittau   Petr Marchenko   Mark Handley   Brad Karp

University College London

Abstract

Software vulnerabilities and bugs persist, and so exploits continue to cause significant damage, particularly by divulging users' sensitive data to miscreants. Yet the vast majority of networked applications remain monolithically structured, in stark contravention of the ideal of least-privilege partitioning. Like others before us, we believe this state of affairs continues because today's operating systems offer isolation primitives that are cumbersome. We present *Wedge,* a system well suited to the splitting of complex, legacy, monolithic applications into fine-grained, least-privilege compartments. Wedge consists of two synergistic parts: OS primitives that create compartments with *default-deny* semantics, which force the programmer to make compartments' privileges explicit; and *Crowbar,* a pair of run-time analysis tools that assist the programmer in determining which code needs which privileges for which memory objects. By implementing the Wedge system atop Linux, and applying it to the SSL-enabled Apache web server and the OpenSSH login server, we demonstrate that Wedge allows fine-grained compartmentalization of applications to prevent the leakage of sensitive data, at acceptable performance cost. We further show that Wedge is powerful enough to prevent a subtle man-in-the-middle attack that succeeds on a more coarsely privilege-separated Apache web server.

1   Introduction

In the era of ubiquitous network connectivity, the consequences of vulnerabilities in server software grow ever more serious. The *principle of least privilege* [16] entails dividing the code into compartments, each of which executes with the minimum privileges needed to complete its task. Such an approach not only limits the harm malicious injected code may cause, but can also prevent bugs from accidentally leaking sensitive information.

A programmer frequently has a good idea which data manipulated by his code is sensitive, and a similarly good idea 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 [2, 6], 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.

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. 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. 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 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.

In this paper, we present *Wedge,* a system that provides programming primitives to 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, Wedge offers 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.

Because a compartment within a complex, legacy, monolithic application may require privileges for many memory objects, Wedge importantly includes *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.

Neither the primitives nor the tools alone are sufficient. Default-deny compartments demand tools that make it feasible for the programmer to identify the mem-

ory objects used by a piece of code, so that he can explicitly enumerate the correct privileges for that code's compartment. And 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 we believe will yield a system that provides fine-grained isolation, and yet is readily usable by programmers.

To demonstrate that Wedge allows fine-grained separation of privileges in complex, legacy, monolithic applications, we apply the system to the SSL-enabled Apache web server and the OpenSSH remote login server for Linux. Using these two applications, we demonstrate that Wedge 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, while still offering acceptable application performance. We further show how the fine-grained privileges Wedge supports can protect against a more subtle attack that combines man-in-the middle interposition and an exploit of the SSL web server.

2 A Motivating Example

To make least-privilege partitioning a bit more concrete, consider how one might partition a POP3 server, as depicted in Figure 1. 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 retrieval 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.

Because of this partitioning, an exploit within the client handler cannot reveal any passwords or e-mails, since it has no access to them. Authentication cannot be skipped since the e-mail retriever will only read e-mails of the user id specified in *uid*, and this can only be set by the login component. One must, however, ensure that code running in the privileged compartments cannot be exploited. This task is simplified since the code that requires audit has been greatly reduced in size by factoring out the client handler, which most likely consists of the bulk of the code. A typical monolithic implementation would combine the code from all three compartments into a single process. An exploit anywhere in the code
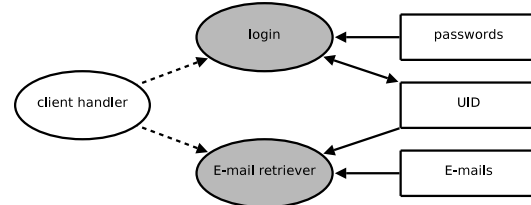
Figure 1: A partitioned POP3 server. Ovals represent code segments and those shaded are privileged. Dashed arrows between ovals indicate the capability of invoking a privileged code segment. Boxes represent memory regions and a one-way arrow from memory to code indicates read permission; two-way arrows indicate read-write.

could cause anything in the process's memory, including passwords and e-mails, to be leaked. Hence, partitioning can reduce the impact of exploits.

Our aim in building Wedge 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.

Wedge achieves this goal with three isolation primitives, which naturally apply to the POP3 example:

Sthreads   An sthread defines a compartment within an application. The programmer assigns access rights to memory and other resources per-sthread. In Figure 1, the unshaded oval is an sthread.

Tagged Memory   When allocating memory, the programmer may mark that memory with a single tag. Access rights to memory are granted to sthreads in terms of these tags (*e.g.,* "read/write for memory with tag t"). In Figure 1, each rectangle is a memory region with a distinct tag.

Callgates   A callgate executes code with different privileges than its caller. An sthread typically runs with the least privilege possible. When it must perform an operation requiring enhanced privilege, it invokes a callgate that performs the operation on its behalf. A callgate defines a narrow interface to privileged code and any sensitive data it manipulates, and thus allows improved adherence to least-privilege partitioning. In Figure 1, each shaded oval is a callgate.

3 System Design

We now describe the operating system primitives that Wedge introduces, and thereafter, the Crowbar run-time analysis tools that ease programmer use of these primitives.

3.1 Sthreads

An sthread is the embodiment of a compartment in a partitioned application. It consists of a thread of control and an associated security policy that specifies:

- The memory tags the sthread may access, and the permissions for each (read, read-write, copy-on-write).

- The file descriptors the sthread may access, and the permissions for each (read, write, read-write).

- The callgates the sthread may invoke.

- A UNIX user id, root directory, and an SELinux policy [10], which limits the system calls that may be invoked.

A newly created sthread holds no access rights by default, apart from copy-on-write access to a pristine snapshot of the original parent process's memory, taken just *before* the execution of *main*, including the sthread's (as yet unused) private stack and heap. (We explain further in Section 4.1 why this snapshot does not contain sensitive data, and is essential for correct execution.) It is unable to access any other memory nor any file descriptor from its creator, nor invoke any callgates or system calls. Note that all system calls retain the standard in-kernel privilege checks, based on the caller's user id, root directory, and SELinux policy. A parent may grant a child sthread access to its resources simply by attaching an appropriate security policy to the child sthread when creating it.

An sthread can only create a child sthread with equal or lesser privileges than its own. Specifically, a parent can only grant a child access to subsets of its memory tags, file descriptors, and authorized callgates. Similarly, the userid and filesystem root of a child sthread can be changed only according to UNIX semantics (*i.e.,* only if the parent runs as superuser), and any changes in the SELinux policy must be explicitly allowed as domain transitions in the system-wide SELinux policy.

Because most CPUs do not support write-only memory permissions, Wedge does not allow them; the programmer must instead grant read-write permissions.

3.2 Tagged Memory

The programmer expresses memory privileges for sthreads in terms of *tags*, which are assigned to memory regions at allocation time. When he wishes to share tagged memory, he grants privileges for that tag to a newly created sthread. The tag namespace is flat, so privileges for one tag never imply privileges for other tags.

Programmers allocate tagged memory in two steps. First, the programmer must create a tag. This operation essentially allocates a memory segment and stores a mapping from the tag to the segment. Next, the programmer invokes a tagged memory allocation (*smalloc*):

he specifies a tag and a desired size, and thus allocates a buffer of that size from the segment with that tag. The ability to identify many memory regions with a single tag simplifies policy specification. Note that any memory an sthread allocates without using this two-stage process (tag, then *smalloc*) has no tag, and thus cannot be accessed by other sthreads: it cannot even be named in a security policy. In this way, computations performed on an sthread's stack or heap are by default strongly isolated from other sthreads. We use the standard hardware page protection mechanism to enforce the access permissions for tagged memory specified in an sthread's security policy.

When writing new applications, the mechanisms described so far suffice for tagging memory. But when partitioning existing applications, one may need to tag global variables, or convert many *malloc* calls within a function to use *smalloc* instead, which may not even be possible for allocations in binary-only libraries. We therefore provide two additional mechanisms for tagging memory, specifically tailored for splitting legacy applications. The first allows declaring globals with a tag, and works by placing all globals with the same tag in a distinct, contiguous section of the ELF binary. The second allows the programmer to specify that *all* calls to the standard C *malloc* between two points in a program should be replaced with *smalloc* calls with a particular tag. To do so, the programmer simply places calls to utility functions *smalloc_on* and *smalloc_off* at the desired start and end points, respectively. These aids greatly simplify the introduction of compartments; much of the work of partitioning consists of identifying and tagging memory correctly.

3.3 Callgates

A callgate is a portion of code that runs with different (typically higher) privileges than its caller. In our POP3 example in Figure 1, the login and e-mail retriever entities are callgates. Instantiating a callgate with access to sensitive data isolates the sensitive data from access by untrusted code, such as the sthread that parses network input in the POP3 server.

A callgate is defined by an entry point, a set of permissions, and a trusted argument supplied by the callgate's creator (usually a pointer into trusted memory), that the kernel will pass to the entry point when the callgate is invoked. The trusted argument allows the callgate's creator to pass the callgate input that cannot be tampered with by its caller. A callgate also inherits the filesystem root and user id of its creator. A callgate's permissions must be a subset of those of the sthread that creates the callgate. After a privileged sthread creates a callgate, it may spawn a child sthread with reduced privilege, but grant that child permission to invoke the callgate. Upon callgate invoca-

tion, a new sthread with the callgate's permissions is created, and begins execution at the entry point specified by the callgate's original creator. The caller blocks until the callgate terminates, and then collects any return values.

The dominant costs of invoking a short-lived callgate are those incurred creating and destroying the underlying sthread. For throughput-critical applications, we provide long-lived *recycled callgates*, which amortize their creation cost over many invocations. Because they are reused, recycled callgates do trade some isolation for performance, and must be used carefully; should a recycled callgate be exploited, and called by sthreads acting on behalf of different principals, sensitive arguments from one caller may become visible to another.

3.4 Crowbar: Partitioning Assistance

To identify the many memory dependencies between different blocks of code, and hence make default-deny partitioning primitives usable in practice, we provide *Crowbar,* a pair of Linux tools that assist the programmer in applying the primitives in applications. Broadly speaking, these tools analyze the run-time behavior of a legacy monolithic application to identify exactly which items in memory are used by which specific pieces of code, with what modes of access, and where all those items were allocated. This analysis suggests a set of privileges that appear required by a particular piece of code.

The programmer uses Crowbar in two phases. In Crowbar's run-time instrumentation phase, the *cb-log* tool logs memory allocations and accesses made by the target application. In Crowbar's analysis phase, the programmer uses the *cb-analyze* tool to query the log for complex memory access patterns during the application's run that are relevant when partitioning the application in accordance with least privilege.

First, *cb-log* produces a trace of all memory accesses. *cb-log* stores a complete backtrace for every memory read or write during execution, so that the programmer can determine the context of each access. These backtraces include function names and source filenames and line numbers. *cb-log* identifies global memory accesses by variable name and source code location; stack memory accesses by the name of the function in whose stack frame the access falls; and heap memory accesses by a full backtrace for the original *malloc* where the accessed memory was first allocated. This information helps programmers identify which globals to tag using our mechanisms, which stack allocations to convert to heap ones, and which specific *malloc* calls (revealed in our trace) to convert to *smalloc* calls.

Second, after *cb-log* produces a trace, the programmer uses *cb-analyze* to query it for specific, summarized information. The supported queries are:

- Given a procedure, what memory items do it *and all its descendants in the execution call graph* access during their execution, and with what modes of access? When the programmer wishes to execute a procedure in a least-privilege sthread, he uses this query to learn the memory items to which he must grant that sthread access, and with what permissions.

- Given a list of data items, which procedures use any of them? When the programmer wishes to create a callgate with elevated privileges to access sensitive data, he uses this query to learn which procedures should execute within the callgate.

- Given a procedure known to generate sensitive data, where do it and *all its descendants in the execution call graph* write data? When the programmer wishes to learn which data may warrant protection with callgates, he uses this query to identify the memory that should be kept private to that callgate. This query is particularly useful in cases where a single procedure (and its children) may generate large volumes of sensitive data; it produces data items of interest for queries of the previous type.

We stress that for the first and third query types, including children in the execution call graph makes *cb-analyze* particularly powerful; one often need not understand the complex call graph beneath a procedure to partition an application.

Crowbar is useful even after the initial partitioning effort. For example, after code refactoring, an sthread may stop functioning, as it may access additional memory regions not initially specified in its policy. We provide an *sthread emulation* library, which grants sthreads access to *all* memory, so that protection violations do not terminate sthreads. The programmer may use this library with Crowbar to learn of all protection violations that occur during a complete program execution.

Because Crowbar is trace-driven, the programmer will only obtain the memory permissions used during one particular run. To ensure coverage of as broad a portion of the application as possible, the programmer may generate traces by running the application on diverse innocuous workloads with *cb-log*, and running *cb-analyze* on the aggregation of these traces.

4 Implementation

Wedge's implementation consists of two parts: the OS isolation primitives, for Linux kernel version 2.6.19, and the userland Crowbar tools, *cb-log* and *cb-analyze*.

4.1 Isolation Primitives

Table 1 shows Wedge's programming interface.

Sthread-related calls	
int sthread_create(sthread_t *thrd, sc_t *sc, cb_t cb, void *arg); int sthread_join(sthread_t thrd, void **ret);	

Memory-related calls	
tag_t tag_new();	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 *arg); void* cgate(cg_t cb, sc_t *perms, void *arg);	

Table 1: The Wedge programming interface.

Sthread-related Calls The programming interface for sthreads closely resembles that for pthreads, apart from the introduction of a security policy argument (*sc*). We implement sthreads as a variant of Linux processes. Rather than inheriting the entire memory map and all file descriptors from its parent, a newly spawned sthread inherits only those memory regions and file descriptors specified in the security policy. As with *fork*, the new sthread will have its own private signal handlers and file descriptor copies, so receiving a signal, closing a file descriptor, and exiting do not affect the parent.

Sthreads also receive access to a private stack and a private copy of global data. The latter represents the memory map of the application's first-executed process, just before the calling of the C entry point *main*. This memory is vital to sthread execution, as it contains initialized state for shared libraries and the dynamic loader. It does not typically, however, contain any sensitive data, since the application's code has yet to execute. In cases where statically initialized global variables are sensitive, we provide a mechanism (*BOUNDARY_VAR*) for tagging these, so that sthreads do not obtain access to them by default. Our implementation stores a copy of the pages of the program just before *main* is called, and marks these pages copy-on-write upon *main*'s invocation or any sthread creation.

Memory-related Calls *smalloc* and *sfree* mimic the usual *malloc* and *free*, except that *smalloc* requires specification of the tag with which the memory should be allocated. Tags are created using *tag_new*, a system call that behaves like anonymous *mmap*. Unlike *mmap*, *tag_new* does not merge neighboring mappings, as they may be used in different security contexts. Apart from creating a new memory area, *tag_new* also initializes internal bookkeeping structures used by *smalloc* and *sfree* on memory with that tag. The *smalloc* implementation is derived from *dlmalloc* [9].

Much of *tag_new*'s overhead comes from system call overhead and initializing the *smalloc* bookkeeping struc-

tures for that tag. We mitigate system call overhead by caching a free-list of previously deleted tags (*i.e.,* memory regions) in userland, and reusing them if possible, hence avoiding the system call. To provide secrecy, we scrub a tag's memory contents upon tag reuse. Rather than scrubbing with (say) zeros, we copy cached, pre-initialized *smalloc* bookkeeping structures into it, and thus avoid the overhead of recomputing these contents.

As described in Section 3.2, *smalloc_on* and *smalloc_off* ease the tagging of heap memory. They convert any standard *malloc* which occurs between them into an *smalloc* with the *tag* indicated in *smalloc_on*. To implement this feature, Wedge intercepts calls to *malloc* and *free* using LD_PRELOAD, and checks the state of a global flag indicating whether *smalloc_on* is active; if so, *smalloc* is invoked, and if not, *malloc* is. In our current implementation, this flag is a single per-sthread variable. Thus, *smalloc_on* will not work if invoked recursively, and is neither signal- nor thread-safe. In practice, however, these constraints are not limiting. The programmer can easily save and restore the *smalloc_on* state at the start and end of a signal handler. Should a programmer need to use *smalloc_on* in recursive or thread-concurrent code (within the same sthread), he can easily save-and-restore or lock the *smalloc_on* state, respectively.

The *BOUNDARY_VAR* macro supports tagging of globals, by allowing each global declaration to include an integer *ID,* and placing all globals declared with the same *ID* in the same, separate, page-aligned section in the ELF binary. This allows for specific pages, in this case global variables, to be carved out of the data segment, if necessary. At runtime, the *BOUNDARY_TAG* macro allocates and returns a unique tag for each such *ID*, which the programmer can use to grant sthreads access to globals instantiated with *BOUNDARY_VAR*. This mechanism can be used to protect sensitive data that is statically initialized, or simply to share global data structures between sthreads.

Policy-related Calls These calls manipulate an *sc_t* structure, which contains an sthread policy; they are used to specify permissions for accessing memory and file descriptors. To attach an SELinux policy to an sthread, one specifies the SID in the form of *user:role:type* with *sc_sel_context.*

Callgate-related Calls *sc_cgate_add* adds permission to invoke a callgate at entry point *cgate* with permissions *cgsc* and trusted argument *arg* to a security policy. The callgate entry point, permissions and trusted argument are stored in the kernel, so that the user may not tamper with them, and are retrieved upon callgate invocation. When a parent adds permission to invoke a callgate to a security policy, that callgate is implicitly instantiated

when the parent binds that security policy to a newly created sthread.

To invoke a callgate, an sthread uses the *cgate* call, giving additional permissions *perms* and an argument *arg*. This argument will normally be created using *smalloc*, and the additional permissions are necessary so that the callgate may read it, as by default it cannot. Upon callgate invocation, the kernel checks that the specified entry point is valid, and that the sthread has permission to invoke the callgate. The permissions and trusted argument are retrieved from the kernel, and the kernel validates that the argument-accessing permissions are a subset of the sthread's current permissions.

Callgates are implemented as separate sthreads so that the caller cannot tamper with the callee (and vice-versa). Upon invocation, the calling sthread will block until the callgate's termination. Because the callgate runs using a different memory map (as a different sthread), the caller cannot exploit it by, *e.g.,* invalidating dynamic loader relocations to point to shellcode, or by other similar attacks. Any signals delivered during the callgate's execution will be handled by the callgate. A caller may only influence a callgate through the callgate's untrusted argument.

We currently implement recycled callgates directly as long-lived sthreads. To invoke a recycled callgate, one copies arguments to memory shared between the caller and underlying sthread, wakes the sthread through a futex [3], and waits on a futex for the sthread to indicate completion.

4.2 Crowbar

Crowbar consists of two parts: *cb-log* traces memory access behavior at run-time, and *cb-analyze* queries the trace for summarized data. We only describe *cb-log*, as *cb-analyze* is essentially a text-search tool, and is thus straightforward. *cb-log* uses Pin [11]'s run-time instrumentation functionality. *cb-log* has two main tasks: tracking the current backtrace, and determining the original allocation site for each memory access.

To compute the backtrace, we instrument every function entry and exit point, and walk the saved frame pointers and return addresses on the stack, much as any debugger does. We thus rely on the code's compilation with frame pointers.

To identify original memory allocations, we instrument memory loads and stores. We also keep a list of *segments* (base and limit), and determine whether a memory access lies within a certain segment, and report the segment if so. There are three types of segments: globals, heap, and stack. For globals, we use debugging symbols to obtain the base and limit of each variable. For the heap, we instrument every *malloc* and *free*, and create a segment for each allocated buffer. For the stack, we use a

function's stack frame as the segment. Upon access, together with the segment name, we also log the offset being accessed within the segment. This offset allows the programmer to calculate and determine the member of a global or heap structure being accessed, or the variable within a stack frame being touched.

Our implementation handles *fork* and *pthreads* correctly; it clones the memory map and keeps separate backtraces for the former, and keeps a single memory map but different backtraces for the latter. *cb-log* also supports the sthread emulation library, by logging any memory accesses by an sthread for which insufficient permissions would normally have caused a protection violation. Because the sthread emulation library works by replacing sthreads by standard pthreads, our current implementation does not yet support copy-on-write memory permissions for emulated sthreads.

5 Applications

To validate the utility of Wedge, we apply it to introduce fine-grained, reduced-privilege compartments into two applications: the Apache/OpenSSL web server, and the OpenSSH remote login server. Because SELinux already provides a mechanism to limit system call privileges for sthreads, we focus instead on memory privileges in this paper. Thus, when we partition these two applications, we specify SELinux policies for all sthreads that explicitly grant access to *all* system calls.

5.1 Apache/OpenSSL

Our end-to-end goal in introducing compartments into Apache/OpenSSL is to preserve the confidentiality and integrity of SSL connections—that is, to prevent one user from obtaining the cleartext sent over another user's SSL connection, or from injecting content into another user's SSL connection.

We consider two threat models. In the first, simpler one, the attacker can eavesdrop on entire SSL connections, and can exploit any unprivileged compartment in the Apache/OpenSSL server. In the second, subtler one, the attacker can additionally interpose himself as a man-in-the-middle between an innocent client and the server. Let us consider each in turn. In what follows, we only discuss the SSL handshake as performed with the RSA cipher, but we expect the defenses we describe apply equally well to SSL handshakes with other ciphers.

5.1.1 Simple model (no interposition)

Perhaps the most straightforward isolation goal for Apache/OpenSSL is to protect the server's RSA private key from disclosure to an attacker; holding this key would allow the attacker to recover the session key for any eavesdropped session, past or future. (We presume here that ephemeral, per-connection RSA keys, which
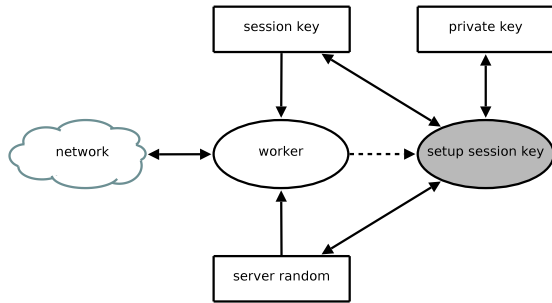
Figure 2: Partitioning to protect against disclosure of private key and arbitrary session key generation.

Figure 3: Man-in-the-middle defense: top-level compartmentalization of Apache.

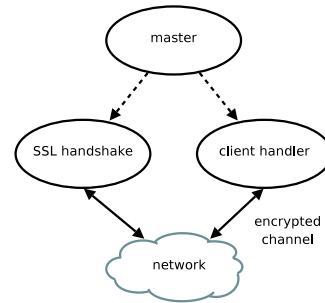provide forward secrecy, are not in use—they are rarely used in practice because of their high computational cost.)

We must also prevent an attacker from *using* the RSA private key to decrypt ciphertext of his choosing, and learning the resulting cleartext; such a decryption oracle would serve the attacker equally well for recovering session keys for past or future sessions.

Once the RSA private key is put out of reach for the attacker, how else might he recover a session key that matches one used in an eavesdropped connection? Note that the server *must* include code that generates SSL session keys—if it does not, it cannot complete SSL handshakes. If the attacker can somehow *influence* this code so as to force the server to generate the same session key that was used for a past connection he has eavesdropped, and learn the resulting session key, he will still achieve his goal.

The SSL session key derives from three inputs that traverse the network: random values supplied by the server and client, both sent in clear over the network during the SSL handshake, and another random value supplied by the client, sent over the network encrypted with the server's public key [15]. Note that by eavesdropping, the attacker learns all three of these values (the last in ciphertext form).

We observe that it is eminently possible to prevent an attacker who exploits the server from usefully influencing the output of the server's session key generation code. In particular, we may deny the network-facing compartment of the server the privilege to dictate the server's random contribution to the session key. Instead, a privileged compartment, isolated from the unprivileged one, may supply the server's random contribution (which is, after all, generated by the server itself).

To meet the above-stated goals, we partitioned Apache 1.3.19 (OpenSSL 0.9.6) as shown in Figure 2. We create one *worker* sthread per connection, which encapsulates unprivileged code. This sthread terminates after serving a single request, to isolate successive requests from one another. We allocate the RSA private key in tagged mem-

ory. Although the *worker* has no direct access to the private key, it can still complete the SSL handshake and establish a session key via the *setup session key* callgate. This callgate returns the established session key, which does not yield any information regarding the private key. Thus, the only way to obtain the private key is to exploit the callgate itself; we have thus reduced the application's trusted code base with respect to the private key to the callgate's contents. In order to prevent an attacker who exploits the *worker* from influencing session key generation, we ensure that the *setup session key* callgate itself generates the *server random* input to session key generation, rather than accepting this input as an argument from the *worker*. Because the session key is a cryptographic hash over three inputs, one of which is random from the attacker's perspective, he cannot usefully influence the generated session key.

5.1.2 Containing man-in-the-middle attacks

We now consider the man-in-the-middle threat model, wherein the attacker interposes himself between a legitimate client and the server, and can eavesdrop on, forward, and inject messages between them. First, observe that the partitioning described for the previous threat model does not protect a legitimate client's session key in this stronger model. If the attacker exploits the server's *worker*, and then passively passes messages as-is between the client and server, then this compromised *worker* will run the attacker's injected code during the SSL handshake with the legitimate client. The attacker may then allow the legitimate client to complete the SSL handshake, and leak the legitimate client's session key (readable by the *worker*) to the attacker. The defense against this subtler attack, as one might expect, is finer-grained partitioning of Apache/OpenSSL with Wedge.

Recall that there are two phases to an SSL session. The first phase consists of the SSL handshake and authentication. After this first phase, the client has authenticated the server, by verifying that the server can decrypt random data encrypted with the server's public key. In addition, the client and server have agreed on a session key
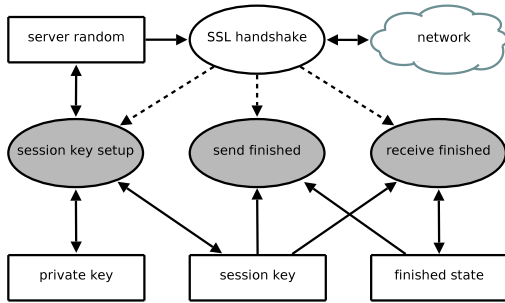
Figure 4: SSL handshake compartmentalization.

to use as the basis for an encrypted and MAC'ed channel between them. In the second phase, application requests and responses traverse this channel.

For privilege separation, there are different threat models for the two phases, so we treat them separately. Figure 3 shows the high-level partitioning needed to implement this two-phase approach. The master exists purely to start and stop two sthreads, and enforce that they only execute sequentially. The *SSL handshake* sthread handles the first phase. This sthread must be able to read and write cleartext to the network to perform the handshake, and while it must be able to call callgates to generate the session key, it should not have access to the resulting key. As this cleartext-reading, network-facing sthread may be exploited, its privileges are limited to those needed for the handshake.

Once the *SSL handshake* sthread completes the handshake, it terminates. The master, which waits for this sthread to terminate, only then starts the *client handler* sthread to handle the second phase. If the attacker attempts to mount a man-in-the-middle attack during the first phase, there are only two possible outcomes: either the handshake fails to complete, and no harm is done, or it does complete, and generates a session key that is not known to the attacker. The *client handler* sthread, however, *does* have access to this session key, and thus it is connected via the SSL-encrypted and -MAC'ed channel to the legitimate client. Despite the man-in-the-middle attack during the first phase, the attacker ends up on the outside of this protected channel.

Man-in-the-middle attacks during the second phase are much more difficult for the attacker, because of the MAC protection on the channel. Data injected by the attacker will be rejected by the *client handler* sthread, and not reach further application code. The attacker's only recourse now is to attempt to exploit the symmetric key decryption code itself. This code is much simpler to audit for vulnerabilities, but as we shall show, further partitioning is also possible to provide defense in depth. We now describe the details of these two phases.

First stage: SSL Handshake   Figure 4 shows the partitioning for the *SSL handshake* stage. The *private key* memory region contains the server's private key, and the *session key* memory region stores the session key. The network-facing *SSL handshake* sthread coordinates the SSL handshake and establishes the session key, *without* being able to read or write it directly; as shown in Figure 4, *SSL handshake* holds neither read nor write permissions for the *session key* tagged memory region. Nevertheless, during the SSL handshake protocol, the server must decrypt one message and encrypt one message with the session key. *SSL handshake* cannot be permitted to invoke callgates that simply encrypt or decrypt their arguments, either; if *SSL handshake* were exploited, the attacker could use them as encryption and decryption oracles, to decrypt ciphertext from the legitimate client.

To understand how one may partition the server to deny *SSL handshake* access to an encryption or decryption oracle for the session key, yet still allow the uses of the session key required by the SSL handshake, one must examine more closely how the SSL handshake protocol uses the session key. After the server and client agree on the session key, they exchange session-key-encrypted SSL Finish messages, to verify that both parties agree on the content of all prior messages exchanged during the handshake. Each SSL Finish message includes a hash derived from all prior messages sent and received by its sender.

We instantiate two callgates, both of which *SSL handshake* may invoke: *receive finished*, which processes the client's SSL Finish, and *send finished*, which generates the server's SSL Finish. *Receive finished* takes a hash derived from all past messages and the client's SSL Finished message as arguments, and must decrypt the SSL Finished message in order to verify it. Once verification is complete, *receive finished* hashes the resulting cleartext together with the hash of past messages, to prepare the payload of the server's SSL Finished message. It stores this result in *finished state,* tagged memory accessible only to the *receive finished* and *send finished* callgates. The only return value seen by *SSL handshake* is a binary success/failure indication for the validation of the client's SSL Finished message. Thus, if *SSL handshake* is exploited, and passes ciphertext from the innocent client to *receive finished* in place of an SSL Finished message, *receive finished* will not reveal the ciphertext.

Send finished simply uses the content of *finished state* to prepare the server's SSL Finished message, and takes no arguments from *SSL handshake*. Data does flow from *SSL handshake* into *finished state*, via *receive finished*. But as *receive finished* hashes this data, an attacker who has exploited *SSL handshake* cannot choose the input that *send finished* encrypts, by the hash function's non-invertibility.
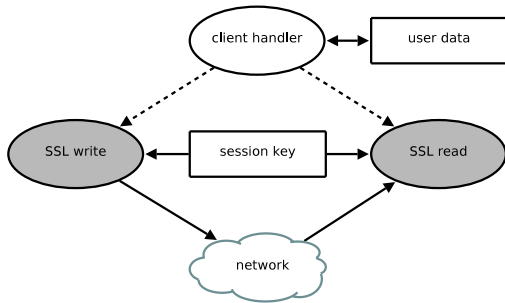
Figure 5: Client handler compartmentalization.

We conclude that if the attacker exploits the *SSL handshake* sthread, he will have no direct access to the session key and, equally important, no access to an encryption or decryption oracle for the session key.

Our implementation fully supports SSL session caching, which varies the final message exchanges in the SSL handshake slightly; we omit those details in the interest of brevity.

Second stage: SSL Client Handler   After *SSL handshake* completes the handshake, it exits. The *master*, previously awaiting this event, then starts *client handler* to process the client's requests. (If *SSL handshake* is exploited, and does not exit, the *master* will not start the *client handler*.)

Figure 5 depicts the compartmentalization of the *client handler* phase. Here we use two callgates, *SSL read* and *SSL write*, to perform encryption and decryption operations using the session key, respectively. Since *client handler* does not have direct access to the network, an attacker must inject correctly MAC'ed and encrypted ciphertext into the user's connection to compromise *client handler*; otherwise, the injected messages will be dropped by *SSL read* (because the MAC will fail). Because of the partitioning of *SSL handshake,* though, the attacker cannot learn the session key, which includes the MAC key. And thus, the attacker will not be able to exploit *client handler*, and so won't be able to leak users' cleartext data (stored in the *user data* memory region).

One unique aspect of the partitioning in Figure 5 is that the unprivileged *client handler* sthread does not have write access to the network. This design choice is an instance of defense-in-depth. In the extremely unlikely event that the attacker manages to exploit *SSL read*, he cannot directly leak the session key or user data to the network, as *SSL read* does not have write access to the network. If, however, the attacker next exploits *client handler* by passing it a maliciously constructed return value, he will still have no direct network access. He will only be able to leak sensitive data as ciphertext encrypted by *SSL write*, and thus only over a covert channel, such as by modulating it over the time intervals between sent packets.

Partitioning Metrics   A partitioning's value is greatest when the greatest fraction of code that processes network-derived input executes in sthreads, and the least fraction in callgates. The latter (man-in-the-middle) partitioning of Apache/OpenSSL we've described contains ≈16K lines of C code that execute in callgates, and ≈45K lines of C code that execute in sthreads, including comments and empty lines. As all code in either category ran as privileged originally, this partitioning reduces the quantity of trusted, network-facing code in Apache/OpenSSL by just under two-thirds. To implement the partitioning, we made changes to ≈1700 lines of code, which comprise only 0.5% of the total Apache/OpenSSL code base.

We note in closing that we relied heavily on Crowbar during our partitioning of Apache/OpenSSL. For example, enforcing a boundary between Apache/OpenSSL's worker and master sthreads required identifying 222 heap objects and 389 globals. Missing even one of these results in a protection violation and crash under Wedge's default-deny model. Crowbar greatly eased identifying these memory regions and their allocation sites, and the sthreads that needed permissions for them.

5.2   OpenSSH

OpenSSH provides an interesting test case for Wedge. Not only was it written by security-conscious programmers, but it is also a leading example of process-level privilege separation [13].

The application-dictated goals for partitioning OpenSSH are:

- Minimize the code with access to the server's private key.

- Before authentication, run with minimal privilege, so that exploits are contained.

- After authentication, escalate to full privileges for the authenticated user.

- Prevent bypassing of authentication, even if the minimally privileged code is exploited.

When partitioning OpenSSH, we started from scratch with OpenSSH version 3.1p1, the last version prior to the introduction of privilege separation. Clearly, an unprivileged sthread is a natural fit for the network-facing code during authentication. As sthreads do not inherit memory, there is no need to scrub, as when creating a slave with *fork* in conventional privilege separation. We explicitly give the sthread read access to the server's public key and configuration options, and read/write access to the connection's file descriptor. We further restrict the
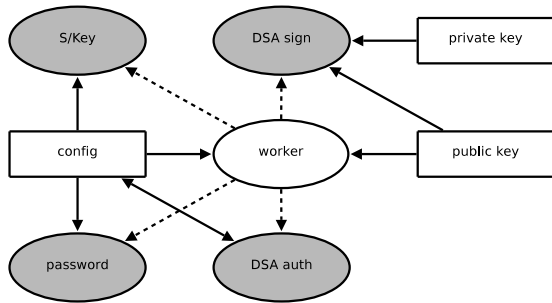
Figure 6: OpenSSH compartmentalization.

sthread by running it as an unprivileged user and setting its filesystem root to an empty directory.

The server's private key is sensitive data, and so must be protected behind a callgate. In our implementation, this callgate contains only 280 lines of C, a small fraction of the total OpenSSH code base, all of which could access the private key in monolithic OpenSSH. To allow authentication, we implemented three more callgates—one each for password, DSA key-based, and S/Key challenge-response authentication. The resulting compartmentalization of OpenSSH appears in Figure 6.

The master sthread (not shown) will spawn one worker for each connection. The worker sthread has no way of tampering with the master, as it cannot write to any memory shared with it. It also cannot tamper with other connections, as sthreads share no memory by default. Thus, all workers (connections) are isolated from each other and from the rest of the OpenSSH daemon.

The worker needs access to the public key in order to reveal its identity to the client. It needs access to the configuration data to supply version strings, supported ciphers, &c. to the client. It has no direct access to sensitive data such as the server's private key or user credentials, and can only make use of these data via callgates.

When the server authenticates itself to the client, it must sign data using its private key. The *DSA sign* callgate takes a data stream as input, and returns its signed hash. The worker cannot sign arbitrary data, and therefore possibly decrypt data, since only the hash computed by the callgate is signed. Exploiting this callgate is the only way that the worker can obtain the private key.

When the user authenticates himself to the server, the *Password*, *DSA auth* or *S/Key* callgate will be invoked, depending on the authentication mechanism negotiated. The password authentication callgate needs access to the configuration data to check whether the user is allowed to login, whether empty passwords are permitted, and whether password authentication is allowed at all. It also needs access to the shadow password file, which is read directly from disk; it has these privileges because it inherits the filesystem root of its creator, not of its caller. The DSA authentication callgate can determine this in-

formation by inspecting the user's allowed keys in the file system, and the S/Key callgate can by checking whether the user has an entry in the S/Key database.

The only way for the worker to change its user ID, and thus effectively let the user log in, is for it to successfully authenticate via a callgate. If the authentication is "skipped" by simply not invoking the callgate, the worker will remain unprivileged. The callgate, upon successful authentication, changes the worker's user ID and filesystem root—an idiom previously applied by Privtrans [1]. Thus, the only way to log in without knowing the user's credentials is to exploit one of the authentication callgates.

We gleaned two important lessons by comparing the Wedge-partitioned OpenSSH with today's privilege-separated OpenSSH. The first concerns the importance of avoiding subtle information leaks from a privileged to an unprivileged compartment. Consider password authentication in (non-Wedge) privilege-separated OpenSSH, which proceeds in two steps. First, the unprivileged slave process sends the username to the privileged monitor process, which either returns NULL if that username does not exist, or the *passwd* structure for that username. Second, the slave sends the password to the monitor, which authenticates the user. The result of the first interaction with the monitor is in fact an information leak—it would allow an exploited slave to invoke the monitor at will to search for valid usernames. This vulnerability remains in today's portable OpenSSH 4.7. The Wedge partitioning keeps OpenSSH's two-step authentication for ease of coding reasons, but the *password* callgate in Figure 6 returns a dummy *passwd* struct (rather than NULL) when the username doesn't exist; this way, even an exploited worker cannot use the *password* callgate to search for usernames. A prior version of OpenSSH contained a similar vulnerability, wherein an S/Key challenge would only be returned if a valid username had been supplied by the remote client [14]. In this case, the OpenSSH monitor and slave leak sensitive information directly to the network; an attacker needn't exploit the slave.

The second lesson concerns the value of default-deny permissions. A past version of OpenSSH suffered from a vulnerability in which the PAM library (not written by the OpenSSH authors, but called by OpenSSH) kept sensitive information in scratch storage, and did not scrub that storage before returning [8]. If a slave that inherited this memory via *fork* were exploited, it could disclose this data to an attacker. A PAM callgate would not be subject to this vulnerability; any scratch storage allocated with *malloc* within the callgate would be inaccessible by the *worker*.

Partitioning Metrics In the Wedge-partitioned version of OpenSSH, ≈3300 lines of C (including com-

ments and whitespace) execute in callgates, and ≈14K execute in sthreads; as all of these lines of code would have executed in a single, privileged compartment in monolithic OpenSSH, partitioning with Wedge has reduced the quantity of privileged code by over 75%. Achieving this isolation benefit required changes to 564 lines of code, only 2% of the total OpenSSH code base.

6   Performance

In evaluating Wedge's performance, we have three chief aims. First, we validate that Wedge's isolation primitives incur similar costs to those of the isolation and concurrency primitives commonly used in UNIX. Second, we assess whether applications instrumented with the Crowbar development tool generate traces tolerably quickly. Finally, we quantify the performance penalty that fine-grained partitioning with Wedge's primitives incurs for Apache/OpenSSL's throughput and OpenSSH's interactive latency.

All experiments ran on an eight-core 2.66 GHz Intel Xeon machine with 4 GB of RAM, apart from the Apache experiments, which ran on a single-core 2.2GHz AMD Opteron machine with 2 GB of RAM, to ease saturation.

Wedge primitives: Microbenchmarks   To examine the cost of creating and running an sthread, we measured the time elapsed between requesting the creation of an sthread whose code immediately calls *exit* and the continuation of execution in the sthread's parent. This interval includes the time spent on the kernel trap for the sthread creation system call; creating the new sthread's data structures, scheduling the new sthread, executing *exit* in the new sthread, destroying the sthread, and rescheduling the parent sthread (whose timer then stops). We implemented analogous measurements for pthreads, recycled callgates, sthreads, standard callgates, and *fork*. In all these experiments, the originating process was of minimal size.

Figure 7 compares the latencies of these primitives. Sthreads and callgates are of similar cost to *fork*, and recycled callgates are of similar cost to pthread creation. Thus, overall, Wedge's isolation primitives incur similar overhead to familiar isolation and concurrency primitives, but support finer-grained control over privilege.

Callgates perform almost identically to sthreads because they are implemented as separate sthreads. Recycled callgates outperform callgates by a factor of eight, as they reuse an sthread, and thus save the cost of creating a new sthread per callgate invocation. For parents with small address spaces, sthread creation involves similar overhead to that of *fork*. For parents with large page tables, however, we expect sthread creation to be faster than *fork*, because only those entries of the page table and

those file descriptors specified in the security policy are copied for a new sthread; *fork* must always copy these in their entirety. Sthreads are approximately 8x slower than pthreads, which incur minimal creation cost (no resource copying) and low context switch overhead (no TLB flush).

Figure 8 shows the cost of creating tags. Allocating memory from a tagged region using *smalloc* costs roughly the same as standard *malloc*, as the two allocators are substantially the same. The difference in overhead between the two lies in tag creation, which is essentially an *mmap* operation, followed by initialization of *malloc* data structures. We manage to outperform *mmap* by caching and reusing previously deleted tags, as described in Section 4.1. The *tag_new* result shown considers the best case, where reuse is always possible; in this case, the operation is approximately 4x slower than *malloc*. In the worst case, when no reuse is possible, tag creation costs similarly to *mmap*, and is hence 22x slower than malloc. We expect reuse to be common in network applications, as the master typically creates tags on a per-client basis, so new client sthreads can benefit from completing ones. Indeed, this mechanism improved the throughput of our partitioned Apache server by 20%.

Crowbar: Run-time Overhead   Figure 9 shows the elapsed time required to run OpenSSH, Apache, and most of the C-language SPECint2006 benchmark applications under *cb-log* (we omit three of these from the figure in the interest of brevity, as they performed similarly to others). We compare these elapsed times against those required to run each application under Pin with *no* instrumentation (*i.e.,* the added cost of Pin alone), and those required to run the "native" version of each application, without Pin. We do not report performance figures for *cb-analyze*, as it consistently completes in seconds. All experiments were conducted using Pin version 2.3.

All applications we measured under *cb-log* (SPEC benchmarks, OpenSSH, and Apache) produced traces in less than ten minutes, and in a mean time of 76 seconds. For the range of applications we examined, *cb-log*'s instrumentation overhead is tolerable. Trace generation occurs only during the development process, and a single trace reveals much of the memory access behavior needed to partition the application. Absolute completion time is more important than relative slowdown for a development tool. Indeed, obtaining a trace from OpenSSH incurs an average 46x slowdown *vs.* native OpenSSH, (2.4x *vs.* Pin without instrumentation), yet the trace for a single login takes less than four seconds to generate.

Pin instruments each fetched basic block of a program once, and thereafter runs the cached instrumented version. From Figure 9, we see that Pin on average executes the applications we measured approximately 7x
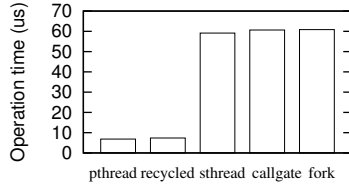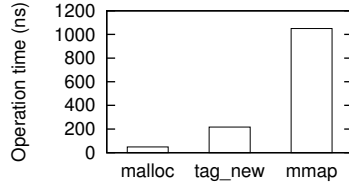
Figure 7: Sthread calls.

Figure 8: Memory calls.

Experiment	Vanilla	Wedge	Recycled
Apache sessions cached (req/s)	1238	238	339
Apache sessions not cached (req/s)	247	132	170
ssh login delay (s)	0.145	0.148	
10MB scp delay (s)	0.376	0.370	

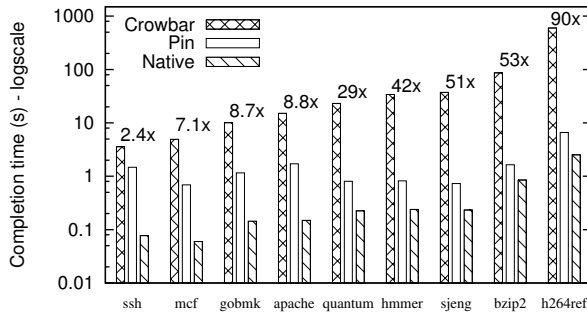**Table 2**: OpenSSH and Apache performance.

Figure 9: Overhead of *cb-log*. The number above an application's bars indicates the ratio between run time under Pin without instrumentation and the run time under Pin with Crowbar.

slower than they execute natively. Note that Pin's overhead is least for applications that execute basic blocks many times, as do many of the SPEC benchmarks. For OpenSSH and Apache, which do not repeatedly execute basic blocks to the extent the SPEC benchmarks do, caching of instrumented code pays less of a dividend.

Cb-log must of course execute added instrumentation instructions in each basic block, and hence it slows applications much more than Pin alone does. On average, applications run 96x more slowly under *cb-log* than they do natively, and 27x more slowly under *cb-log* than they do under Pin with no instrumentation.

Applications: End-to-end Performance The top half of Table 2 shows the maximum throughput that the Wedge-partitioned version of Apache and the original version of Apache can sustain, in requests per second. We consider two Wedge-partitioned implementations: one with standard callgates ("Wedge"), and one with recycled callgates ("Recycled"). We also measured Apache's performance with and without session caching. In these experiments, four client machines request static web pages using SSL, across a 1 Gbps Ethernet.

The isolation benefits Wedge offers Apache come at a performance cost that depends on the client workload. When all client SSL sessions are cached (and thus, the server need not execute the cryptographic operations in the SSL handshake), Wedge-partitioned Apache with recycled sthreads achieves only 27% of the throughput of unpartitioned Apache. When no SSL sessions are cached, however, Wedge-partitioned Apache with recy-

cled sthreads reaches 69% of unpartitioned Apache's throughput. The all-sessions-cached workload is entirely unrealistic; we include it only to show the worst-case overhead that partitioning Apache with Wedge incurs.

Because SSL session caching allows clients to reuse session keys, it eliminates the significant SSL handshake cost that the server incurs once per connection in the non-session-cached workload. Thus, Wedge-induced overhead is a greater fraction of the total computation required when the server accepts a connection for a cached session. This overhead is directly proportional to the number of sthreads and callgates created and invoked per request. Each request creates two sthreads and invokes eight callgates (nine, for non-session-cached clients)—a few callgates are invoked more than once per request. Vanilla Apache instead uses a pool of (reused) workers, so it does not pay process creation overhead on each request; it thus provides no isolation between successive requests executed by the same worker, but is faster. Recycled callgates are an effective optimization—they increase Wedge-partitioned Apache's throughput by 42% and 29% for workloads with and without session caching, respectively.

The bottom half of Table 2 compares the latencies of operations in pre-privilege-separated OpenSSH and in the Wedge-partitioned version, for a single OpenSSH login and for uploading a single 10 MB file using scp. Wedge's primitives introduce negligible latency into the interactive OpenSSH application.

7 Discussion

We now briefly discuss a few of Wedge's limitations, and topics that bear further investigation. Several of these limitations concern callgates. First, we rely on their 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 no 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 call-gates. As this code is of manageable size—less than 2000 lines—we believe that it can be audited. Perhaps more worryingly, we must also trust the remainder of the Linux kernel, though like Flume [7], we also inherit Linux's support for evolving hardware platforms and compatibility with legacy application code.

Crowbar is an aid to the programmer; not a tool that automatically determines a partitioning by taking security decisions on its own. We believe that automation could lead to subtle vulnerabilities, such as those described in Section 5.1.2, that are not apparent from data dependencies alone. 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 aid, and it is the programmer who enforces correct isolation.

Wedge does not deny read access to the text segment; thus, a programmer cannot use the current Wedge implementation to prevent the *code* for an sthread (or indeed, its ancestors) from being leaked. Wedge provides no direct mechanism to prevent DoS attacks, either; an exploited sthread may maliciously consume CPU and memory.

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 execute in practice. Static analysis would report the exhaustive set of permissions 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 leak 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*.

8 Related Work

Many have recognized the practical difficulty of partitioning applications in accordance with least privilege. OKWS [5] showed that with appropriate contortions, UNIX's process primitives can be used to build a web server of compartments with limited privileges. Krohn *et al*. [6] lament that UNIX is more of an opponent than an ally in the undertaking, with the result that programmers create single-compartment, monolithic designs, wherein all code executes with the union of all privilege required across the entire application.

A trio of Decentralized Information Flow Control (DIFC) systems, Asbestos, HiStar, and Flume [2, 7, 18] offer a particularly general approach to fine-grained isolation. DIFC allows untrusted code to observe sensitive data, but without sufficient privilege to disclose that data. Wedge does not provide any such guarantees for untrusted code. In its all-or-nothing privilege model, a compartment is either privileged or unprivileged with respect to a resource, and if a privileged compartment is exploited, the attacker controls that compartment's resources with that compartment's privileges. Our experience thus far suggests that when applying Wedge to legacy, monolithic code, only a small fraction of the original code need run in privileged compartments. The price of DIFC is heightened concern over covert channels, and mechanisms the programmer must employ to attempt to eliminate them [18]—once one allows untrusted code to see sensitive data, one must restrict that code's communication with the outside world. Because Wedge is not a DIFC system, it does not share this characteristic. Wedge still may disclose sensitive information through covert channels, but *only* from compartments that are privileged with respect to sensitive information. DIFC systems leave the daunting challenge of designing an effective partitioning entirely to the programmer. Wedge's Crowbar tools focus largely on helping him solve this complementary problem; we believe that similar tools may prove useful when partitioning applications for Asbestos, HiStar, and Flume, as well.

Jif [12] brings fine-grained control of information flow to the Java programming language. Its decentralized label model directly inspired DIFC primitives for operating systems. Jif uses static analysis to track the propagation of labels on data through a program, and validate whether the program complies with a programmer-supplied information flow control policy. The Wedge isolation primitives do not track the propagation of sensitive data; they use memory tags only to allow the programmer to grant privileges across compartment boundaries. Wedge's Crowbar development tool, however, provides simple, "single-hop" tracking of which functions use which memory objects in legacy C code, whose lack of strong typing and heavy use of pointers make accurate static analysis a challenge. Jif uses static analysis to detect *prohibited* information flows, while Crowbar uses dynamic analysis to reveal to the programmer what are most often *allowed* information flows, to ease development for Wedge's default-deny model.

Jif/split [17] extends labeled information flow for Java to allow automated partitioning of a program across distributed hosts, while preserving confidentiality and integrity across the entire resulting ensemble. Because it allows principals to express which hosts they trust, Jif/split can partition a program such that a code fragment that executes on a host only has access to data owned by principals who trust that host. Wedge targets finer-grained partitioning of an application on a single host. In cases where robustness to information leakage depends on the

detailed semantics of a cryptographic protocol, such as in the defense against man-in-the-middle attacks on SSL in Section 5.1, Jif/split would require careful, manual placement of declassification, based on the same programmer knowledge of the protocol's detailed semantics required when using Wedge's primitives.

Provos proposes privilege separation [13], a special case of least-privilege partitioning targeted specifically for user authentication, in which a monolithic process is split into a privileged monitor and one (or more) unprivileged slaves, which request one of a few fixed operations from the monitor using IPC. Privman [4] is a reusable library for use in implementing privilege-separated applications. Privilege separation allows sharing of state between the monitor and slave(s), but does not assist the programmer in determining what to share; Crowbar addresses this complementary problem. Privtrans [1] uses programmer annotations of sensitive data in a server's source code and static analysis to automatically derive a two-process, privilege-separated version of that server. Wedge arguably requires more programmer effort to use than Privtrans, but also allows much richer partitioning and tighter permissions than these past privilege separation schemes; any number of sthreads and callgates may exist within an application, interconnected in whatever pattern the programmer specifies, and they may share disjoint memory regions with one another at a single-byte granularity.

9 Conclusion

Programmers know that monolithic network applications are vulnerable to leakage or corruption of sensitive information by bugs and remote exploits. Yet they still persist in writing them, because they are far easier to write than carefully least-privilege-partitioned ones. Introducing tightly privileged compartments into a legacy monolithic server is even harder, because memory permissions are implicit in monolithic code, and even a brief fragment of monolithic code often uses numerous scattered memory regions. The Wedge system represents our attempt to exploit the synergy between simple, default-deny isolation primitives on the one hand, and tools to help the programmer reason about the design and implementation of the partitioning, on the other. Our experience applying Wedge to Apache/OpenSSL and OpenSSH suggests that it supports tightly privileged partitioning of legacy networking applications well, at acceptable performance cost. Above all, we are particularly encouraged that Wedge has proven flexible and powerful enough to protect Apache against not only simple key leakage, but also against complex man-in-the-middle attacks.

Wedge is publicly available at:
`http://nrg.cs.ucl.ac.uk/wedge/`

References

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

[2] 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.

[3] H. Franke, R. Russell, and M. Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in Linux. In *Ottawa Linux Symposium*, 2002.

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

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

[6] 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.

[7] 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*, 2007.

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

[9] D. Lea. A memory allocator by Doug Lea. `http://g.oswego.edu/dl/html/malloc.html`.

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

[11] 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.

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

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

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

[15] E. Rescorla. *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley Professional, 2000.

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

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

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