Bodyguard:

Running Protected Applications in Untrusted Operating Systems

by

Russell Lewis

Submitted to the Department of Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

at the

UNIVERSITY OF ARIZONA
Technical Report TR10-03
April 30, 2010

Abstract

In this thesis, we present a method to run an application within a commodity operating system without risking either the correctness or privacy of the application should the operating system be compromised. Using a hypervisor, we invisibly intercept all attempts by the operating system to corrupt the state of the application or access its data. We accomplish this first by tracking the current state of the virtual space and verifying all actions by the operating system which might change this state, and second by replacing the contents of physical pages with randomly generated restorable signatures when the operating system attempts to access the contents. The system is sufficiently flexible to allow a binary-unmodified operating system to perform typical tasks such as copy-on-write, fork(), and swap, and sufficiently automatic that the protected application only needs small modifications. Finally, we present automatic methods for adapting a legacy application which are able to provide complete and seamless protection for many applications.

Contents

Figures

Chapter 1: Introduction

> "Honey, I forgot to duck."
>
> – Ronald Reagan, to his wife, shortly after being shot
>
> An important political figure must interact with the public. While most of the people he meets will have good intentions, a handful may attempt to cause him harm. The job of a bodyguard is to allow appropriate interactions between the politician and the people, while preventing attacks of all kinds.

This thesis presents Bodyguard, a system we have devised that allows a critical application to run on a completely unmodified commodity operating system. Bodyguard monitors the actions of the OS, and allows most operations to complete normally, but it is able to identify and prevent many attacks.

Bodyguard implements this protection with a hypervisor, which runs at a higher authority level than the OS. The hypervisor tracks the working set of a certain number of protected processes, and is able to tell which code in the system is "trusted" code (authorized to access and modify the state of a protected process) and which is untrusted. If untrusted code attempts to do something dangerous, Bodyguard will either modify the action in order to make it safe, or terminate the protected process before corruption can occur.

Remarkably, Bodyguard's protections allow an entirely unmodified operating system to perform seemingly-risky operations, such as changing page tables, moving pages to/from a swap file, and handling COW (copy-on-write). While Bodyguard gives the untrusted OS complete access to the page tables and the contents of all physical frames, it enforces that the internal state of all protected processes are always preserved, and that no private data leaks to untrusted code.

Traditional security systems attempt to prevent an attacker from taking control of the OS, so that the state of critical applications will not be compromised. Bodyguard instead assumes that an attacker has (or might have) control of the operating system from the very start, and implements complete and almost invisible protection despite that fact.

1.1 Signatures

The central concept in Bodyguard is the *signature*, which is a randomly generated page of data that represents the contents of a private page, as it existed at some point in time.

Any time that untrusted code attempts to read or write a private page, Bodyguard ensures that it sees the signature for the page, rather than the private data. The untrusted code may copy the signature to other pages (COW), store it to disk (swap), or use it for any other purpose. Later, when the protected process attempts to access some physical page, Bodyguard expects to finds the signature. If it does, then Bodyguard will automatically convert the signature back to the private data that it represents. Thus, when the OS

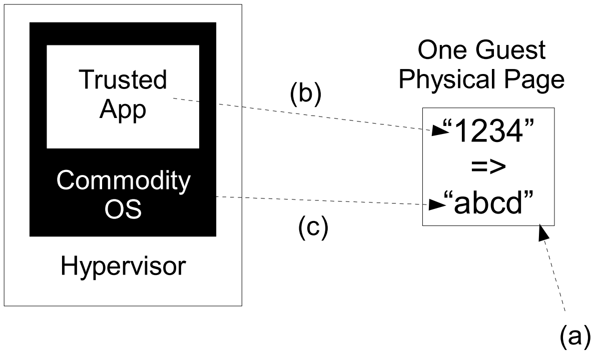


Figure 1: Signatures

(a) The signature "abcd" represents the private data "1234"
(b) The trusted app reads the page, and sees "1234"
(c) The OS reads the same page, and sees "abcd"

copies a signature to some other frame, it is effectively copying the private data, yet doing so in a manner that cannot be used to reveal any secrets.

Signatures also play a major role in preventing an attacker from corrupting the state of a protected process. Any time that Bodyguard is forced to replace a page with a signature, it records the expected contents of that page (the signature). Later, when the protected process attempts to access the page again, Bodyguard confirms that the physical page actually contains the expected signature. If it does not, Bodyguard terminates the process, rather than allow a corrupted access.

Bodyguard uses a similar technique to defend against page table attacks (where the OS might change the page tables to cause a certain virtual page point to incorrect data). Bodyguard keeps track of which physical page currently is used as the backing store for each virtual page. If this changes from one access to the next, Bodyguard converts the old physical page to a signature, and then confirms that the new physical page has the same signature. If it does (as would happen in a valid COW scenario), then the access is permitted. If it does not, then Bodyguard detects an attack and terminates the process.

1.2 Why not just fix corruption?

When corruption is detected, Bodyguard knows what the frame was **supposed** to contain. One possible design alternative would be, rather than to terminate the protected process, to instead fix the frame by writing the correct data to it. Why does Bodyguard terminate the protected process?

First, we believe that once the operating system has done something obviously wrong, it is difficult to "fix" it without breaking some other thing within the OS. What if the page contains critical data for some other program, or for the kernel? Any attempt to force the program to run, despite operating system misbehavior, could have any number of ugly side effects.

Second, we see little point in attempting to continue to run inside an OS known to corrupt protected programs. Even if the hypervisor could safely drive the protected program past this point, it seems quite likely that additional, perhaps more serious, problems may lurk on the horizon. We believe it best to crash the program immediately, which informs the user that this untrusted OS has been compromised and that it should not be used for future work.

1.3 Why not isolate resources?

Another alternative would be to implement each page in a process' working set in hypervisor memory, totally ignoring the OS page tables. Likewise, Bodyguard could implement private networking and private disk. All of the resources could be guaranteed to operate correctly because they are isolated from the untrusted OS. However, in such a scenario, the protected processes are effectively running on the "bare metal" of the hypervisor. If the program is capable of doing so, why is it running inside an untrusted OS at all?

Bodyguard is designed to protect programs that need to run inside a commodity OS because the commodity OS provides compelling features, which are not easily emulated or replicated inside the program itself. Programs that can be adapted to run on the bare metal of the hypervisor do not need Bodyguard's protection. Therefore, Bodyguard assumes that the process must run under the control of the untrusted OS, with the OS allowed to manage paging, CPU allocation, networking, disk, and all other resources. Bodyguard's strategy is to validate the actions that the untrusted OS performs, not to emulate them.

1.4 Thesis Organization

Chapter 2 defines the scope and motivation of this invention. Chapter 3 details the various types of security risks that exist in a typical operating system, as well as several valid situations where an operating system might want to read or write pages that contain private data. It will then sketch a system that could be used to allow the valid accesses while preventing all of the attacks. Chapter 4 discusses Bodyguard's design in detail. Chapter 5 discusses Overshadow, a similar existing system for protecting applications, and compares/contrasts it with our work, discussing tradeoffs and advantages. Chapter 6 details our implementation, including its current limitations. Chapter 7 discusses our results. Chapter 8 concludes with a brief discussion of future work.

Chapter 2: Vulnerabilities in Operating Systems

This chapter specifies the scope of the problem and Bodyguard's solution.

First, it discusses the motivation of this work. Next, it details the types of attacks we consider, including the elements of computing that might be used for an attack. Finally, it details the assumptions, design principles, and general limitations of Bodyguard's protection.

2.1 Motivation

While all programs are designed to run correctly, some are of particular importance. A program may perform critical business functions, deal with classified information, or provide critical safety features. Such programs are carefully designed and thoroughly debugged to maximize quality. Some may be formally analyzed with static analysis tools [1][2][3][4][5], and all are thoroughly empirically tested. The programs are as trustworthy as modern computer science can make them.

However, many of these programs run on commodity operating systems that are far less reliable. Security analysts and hackers regularly find major flaws in operating systems or applications that may give an attacker arbitrary access to the machine. Thus, all the work to perfect an application is (in part) wasted, as a determined attacker is likely to eventually find a flaw in the operating system and use it to disrupt the trusted application.

A variety of strategies exist to combat this problem. First and most importantly, the user must be vigilant about keeping the operating system up-to-date with the latest patches. However, this solution is both incomplete and risky. It is incomplete because there always exists a window of time between when a vulnerability is found and when it is patched; the machine could be compromised in this timeframe [6]. It is risky because patches may sometimes introduce new bugs [7]: either new vulnerabilities may be created, or the implementation of the operating system may change, causing the application to no longer function as designed.

Another alternative is to isolate the machine from the network. This can vary from using firewall software on the computer, to external firewall devices, to actually disconnecting the machine from any network [8]. However, these are only a partial solution [9]. Most firewalls allow outgoing connections (that is, for the machine to access the outside Internet), and client applications have been known to have vulnerabilities. Thus, a machine can be attacked even through a firewall. Moreover, disconnecting the machine entirely is often impractical. In some cases, such as programs that provide services over the Internet, it may be impossible to use a firewall.

Yet another alternative is to run the program on a secure operating system (or hardware platform) that is believed to have fewer vulnerabilities [10][11][12][13][14][15][16][17]. This is problematic for several reasons. First, porting a program to a new operating system generally requires major rewrites of the code, which may be impractical for cost reasons. Such a port may also open new vulnerabilities in the trusted program, if the programmer is not aware of the subtleties of the new environment. Second, a non-standard operating system may actually have *more* vulnerabilities than a commodity operating system, because it has fewer users and less time is spent examining it for flaws [18][19]. Third, a non-standard operating system is likely to lack key features that the program may want to make use of, such as communication protocols, interoperability with other computers, or device drivers. As the non-standard operating system is expanded to match the capabilities of the commodity operating system, it will get more complex, and thus become much more difficult to verify.

A final alternative is to port the operating system itself to a new platform. For instance, one may implement a VMM which provides some sort of security features inside a paravirtualized environment. OSes may then be ported to run on the new VMM. However, this may be difficult or impossible if the operating system is very large, or if the source is unavailable [20]. Moreover, it may be impractical to keep such a port up-to-date with the latest updates of such a system.

2.2 Design Principle: Thin Emulation

This thesis presents a system in which the untrusted OS is expected to perform all of its normal tasks; the hypervisor simply verifies that such tasks are implemented in such a way that the security of the program is not violated. Most application actions are handled automatically by the untrusted OS; a handful of actions must be intercepted and emulated by a very thin emulation layer.

An alternative solution to this problem is to write a complete operating system (or a subset of one) that is linked into existing applications. This operating system provides a backward-compatible interface to the trusted application (so that the trusted application does not have to be ported), but provides an implementation that emulates most operating system functions. For instance, the emulation layer could perform virtual memory page allocation, implement a file system, do timeslicing amongst various processes and threads, etc.

The advantage of such a design is that the hypervisor implements very simplistic and absolute security policy. For instance, it forces a protected process to execute whether or not the untrusted operating system gave it a timeslice; the hypervisor could make it impossible for the untrusted OS to overwrite or read protected physical memory pages or disk blocks; etc.

We reject this design because, in our view, this would require that we re-implement most of the operating system logic. The complexity of the emulation layer would approach the complexity of a full operating system, and thus be just as difficult to verify. Moreover, if the emulation layer implements all of these operations internally, then there is no clear reason why to run the untrusted OS at all. Why not just run the emulation layer directly on a hypervisor?

2.3 The Players

In this thesis, we assume that there are three key players: the *attacker*, *hypervisor*, and *client*. (See Figure 2: The Players.)

We assume that the *attacker* is a malicious person with complete knowledge of the commodity operating system, the protected application, and our protection mechanisms. We will assume that the attacker may have taken control of the operating system arbitrarily long before the client initiates the application, and that the attacker has the ability to control the OS in any way that he desires.

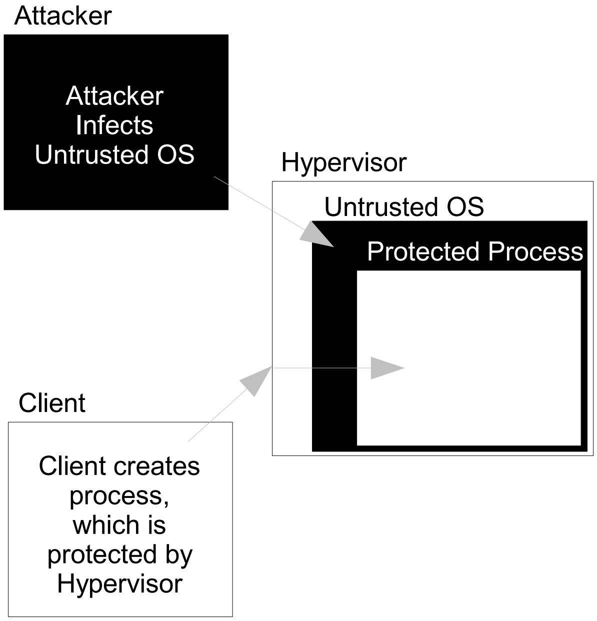

Attacker
Attacker Infects Untrusted OS
Hypervisor
Untrusted OS
Protected Process
Client
Client creates process, which is protected by Hypervisor

Figure 2: The Players

The *hypervisor* is a hardware or software mechanism (probably a VMM) that implements the protections described in this thesis. We assume that it is simple enough that it has no exploitable bugs, and thus that the attacker has no ability to control or snoop on the hypervisor.

Finally, the *client* is an external computer system, which we presume to be **not** corrupted by any attacker. It interacts with the hypervisor, giving it the initial state of new protected processes to run. This client might be a trusted guest OS running inside the same hypervisor as our untrusted OS, or it might be an external computer. We also assume that there is a secure communication channel (presumably ssh or similar) between the client and the hypervisor.

2.4 Attack Classes

This thesis addresses three classes of attack: *snooping, corruption*, and *DOS*.

Snooping attacks involve the attacker accessing private data of the process, perhaps without ever corrupting its state. Bodyguard makes these impossible; any attempt to read private data, by any entity other than the protected application, will return a signature rather than the private data.

Corruption attacks involve changing the process's state in a way that the attacker hopes will not be detected. This could involve corrupting data values in registers or pages, changing the instruction pointer, swapping two or more otherwise-valid data pages or code pages, corrupting the process's initial state, or even causing the application to run code pages generated by the attacker. Bodyguard is designed to prevent all such corruption attacks. When corruption is detected, Bodyguard forces the application to crash, meaning that the attack, once detected, will resolve to a DOS attack, which can be detected as discussed in the previous paragraph.

DOS (Denial of Service) attacks deny some sort of critical resource to a running application. This could involve rejecting requests when the application attempts to allocate resources, failing to provide promised resources (including CPU time), or illegally removing resources (such as causing segfaults on virtual addresses that should be valid). Since an attacker could trivially DOS the application simply by refusing to allow it any CPU time, it is impossible to entirely prevent these attacks. However, we assume that the client will implement timestamps and/or heartbeating to confirm that the protected application is making progress. Other types of DOS attacks should be detectable by the protected application, and reportable to the client using normal mechanisms. Thus, Bodyguard does not have to take any explicit action to detect, prevent, or report DOS attacks.

2.5 Attack Vectors

This thesis provides direct protection for memory and registers. We believe that this is sufficient to provide both correctness and privacy for all other types of resources.

Memory

This thesis presents a method for protecting memory that makes it impossible for the OS to corrupt the application's memory. (More accurately, what it provides is a mechanism to detect corruption, and crash[1] the application before it uses corrupted data.)

[1] Why crash the application? Why not report the error to the application, and allow it to handle the problem as it sees fit? It is certainly possible to notify the application instead of crash. However, remember that the error handling code would need to be protected, just like the rest of the code. It would be possible to hit errors in the error handling code (the classic “double exception” problem). In that circumstance, the hypervisor would have no choice but to crash the application immediately.

This thesis will also present a method for ensuring the privacy of data pages while still allowing the OS to perform ordinary page management tasks, such as swap and copy-on-write.

Registers

This thesis presents a method for protecting registers such that it is impossible for the OS to alter the registers in such a way as to corrupt the application.

This thesis also presents a method for ensuring the privacy of data stored in registers.

IPC

This thesis assumes that the trusted application will encrypt all IPC channels between protected processes so as to prevent both corruption and snooping. Or, as discussed in Future Work, it may emulate IPC using shared memory.

In either case, protection of IPC becomes a matter of protecting the memory and registers that are used to perform the encryption or emulation. Since this can be implemented entirely in the protected process using existing data protection primitives, we consider this to simply be a special case of register and memory protection.

Disk, Network, UI

This thesis does not present a method for protecting disk storage. This thesis assumes that, if the application needs to write to disk, it performs encryption (or some other sufficient security methodology) to ensure the security of what is written.

Likewise, this thesis assumes that the program uses encryption (perhaps via ssh, in a protected child process) to read or write data over the network.

Finally, this thesis assumes that UI is similarly protected. On current operating systems, the only practical UI solution (other than an encrypted command line) is an ssh tunnel to a remote X server. However, one could imagine using a secure UI; this thesis assumes that this program implements its part of any such protocol inside its own process space, or that of a protected child process.

2.6 Memory Model

This thesis depends heavily on a certain memory model, namely the virtual memory model implemented by Linux and other modern operating systems. We acknowledge that other memory models are conceivable, but they are not in common use in modern computing.

In particular, this thesis assumes that:

1) The code and data pages of a process share the same virtual address space.
2) The data pages of a process always have the same contents, when viewed from any block of code in that same process.
3) The address space of a process may contain code and data that are part of the kernel, but such virtual pages always map to different physical pages than application pages (except for the special case of COW pages, which temporarily share a physical page).

Thus, the hypervisor must crash the application in some worst-case circumstances. If some future version of Bodyguard attempts to also implement error handling within the application when possible, that is a valid (but, in our view, not critical) enhancement.

4) The code and data pages of a process never have their contents change, and are never relocated nor removed (in the virtual space)[2], without explicit application action. (Explicit action may involve a write by the application, or a syscall from the application to the kernel.) Overlays (where the same virtual address is used for different code or data, depending on context) are either not used, or are manually managed by the program with mmap/munmap. In particular, there are no automatic mechanisms that magically change overlays without explicit application action.

5) All threads in the same process see the exact same virtual memory contents at all times.

6) When there are multiple processes that are part of the same protected application, memory writes by one process affect the contents of pages in other process if and only if the affected pages are shared pages, such as MAP_SHARED `mmap()`s, `shmat()` regions, etc.

7) The kernel never composes code that the application must run. (During initial loading of an application, or during dynamic linking, the kernel may map in new pages from a file, but the contents precisely reflect the contents of the file and are not modified by the kernel.)

2.7 Limitations

This thesis assumes that the trusted program does not have fundamental errors. For instance, this thesis assumes that the trusted program can accept arbitrary input (verifying it if needed), and respond to any ordinary operations that the operating system might perform (such as sending signals, timeslicing, page swapping, etc.). It must be able to handle any errors that an uncorrupted operating system might deliver, such as disk full or other resource limitations.

Additionally, the trusted program must respond correctly to all possible signals whenever they are not masked off. Bodyguard does not have any way to determine when a signal might or might not be sensible (other than assuming that they cannot be sent while masked off). For example, the attacker might cause SEGV to be sent even though the application has not performed any access which would justify that signal. Thus, the application must include a reasonable default action (which might be to terminate) for all signals which are not masked.

Likewise, the trusted application also must not depend on the delivery of signals for correctness; it must never produce an incorrect result if signals are not delivered, or are delivered out of order.

Finally, if privacy is needed, then the trusted program should not write out any non-encrypted data to disk (including temporary files), network, or any other device.

If the program has fundamental errors, this thesis will not be able to fix them.

2.8 Summary

In this chapter, we specified the scope of the problem and the solution. We discussed the three major players, the three classes of attack, and the various elements of computing vulnerable to attack. We then discussed various design principles, assumptions, and limitations of this work.

[2] The OS may move pages to swap, but the effective virtual contents are not changed, because the OS will swap the page back in before any future access is permitted.

Chapter 3: Security Risks and Protection

This chapter will detail the various types of attacks that are possible, and also several valid operations that the OS might perform and thus Bodyguard must tolerate. Along the way, it will provide a sketch of Bodyguard's protection scheme, and discuss how it operates in when it encounters various attacks or valid operations.

3.1 Corruption Attacks

Corruption attacks take many forms. They may involve overwrites of existing virtual pages, moving pages around inside the virtual space, adding virtual pages that the program didn't ask for, corrupting the initial state of the process (or newly mapped pages), altering the contents of registers (including, but not limited to, the instruction pointer), or corrupting input/output.

3.1.1 Basic Corruption (memory, registers)

There are many different ways in which an attacker might try to corrupt the state of a protected process. These vary from the trivial (overwrite some data) to the complex (swap two valid code or data pages, putting each at the wrong virtual address). Rather than detail every possible strategy, we instead present a general theory for protection.

A protected process only cares about the contents of its registers, and of the pages that it accesses. It doesn't care how the page tables are laid out, which physical pages are used for which virtual pages, etc. It also doesn't care about pages that it never touches. All it cares about are the actual bytes that it accesses.

All corruption attacks, then, have the same form: they alter the contents of a register, or of virtual memory, so that the process reads the wrong data, causing it to perform the wrong action (or to later write the wrong value to another register, or memory). If Bodyguard ensures that the registers always have the correct contents, and that the code and data pages they access likewise have the correct contents at the moment that they are read, then Bodyguard knows that each action that the protected process takes is also correct.

Later in this thesis, we will document how Bodyguard keeps track of whether protected code is running or not, how it protects registers, how it protects virtual page contents, and how it uses these mechanisms to prevent snooping.

3.1.2 Other Miscellaneous Corruption Attacks

The discussion above assumes that there exists some current state of the process (memory and registers) that the hypervisor can track, protect, and update. However, we must also consider attacks that corrupt the initial state of the process, or inputs to it:

- *Bootstrap Attacks:* Bodyguard needs a way to verify the initial state of the process when the entity is first created. This will be detailed in a later section.
- *Mmap() Attacks:* Bodyguard needs a way to verify that newly `mmap()`ed pages have the correct contents. This requires that the program verify the page contents immediately after an `mmap()`. Bodyguard requires that the program use a custom mechanism to confirm its pages,

perhaps by knowing the checksum of all files that it maps. In Future Work, we hope to investigate ways to automate this by creating a special file format that includes checksums within the file itself.

- *Fork():* Bodyguard needs a way to handle `fork()`, and ensure that the newly-created process is an exact duplicate of the old process (but that their pages are COW copies of each other). This will be detailed later.
- *Protected IPC:* Bodyguard needs a way for two protected processes in the same entity to communicate via IPC mechanisms (including pipes) in a way that prevents both corruption of, and snooping on, the private data. Bodyguard assumes that all IPC channels will either be encrypted or (as discussed in Future Work) emulated using shared memory.
- *Client Network Connections:* Likewise, the process may need a network link to its client, through which it may receive inputs and report outputs. Bodyguard assumes that the program will use ssh or similar to protect these connections. Thus, data traveling from the client to the program is protected by ssh while over the network, and then by the protected IPC mechanism above when traveling between the ssh process and other protected processes in that entity.
- *Files on Disk:* The hypervisor does not protect files on disk. If the protected application chooses to write out data to a local disk and later read it back, it must have a way to verify the contents [21]. This can be tied into the same `mmap()` verification mechanism that we mentioned above.
- *IPC and Network to Untrusted Elements:* The program may also include IPC and/or network links to untrusted programs. These will not be protected, obviously, but the program is expected to verify the inputs (if appropriate) before making use of them. The program must realize that an attacker could corrupt input sent into the program, or corrupt the status that it reports out.

3.1.3 Shared Memory Attacks

Shared memory attacks are attacks where the untrusted OS does not properly implement shared memory, either by sharing a physical page between two unrelated virtual pages (allowing a write to one page to modify the other), or by failing to share two virtual pages which should have mapped to the same physical page.

In truth, shared memory attacks are just another form of corruption attack, but we give them special consideration because they require that Bodyguard add a layer of indirection to our protection. (See Figure 3 below.) Bodyguard does not protect **virtual** pages; instead, it protects **logical** pages. A logical page has one or more virtual pages associated with it; a non-shared page has one virtual page, while a shared page can have arbitrarily many virtual pages associated.

When a process attempts to access (execute, read, or write) a virtual page, the hypervisor looks up the virtual → logical association. The logical page stores the information about the current contents of the page; the hypervisor compares the stored contents of the page with the contents of the physical frame, and the access is valid if the contents match.

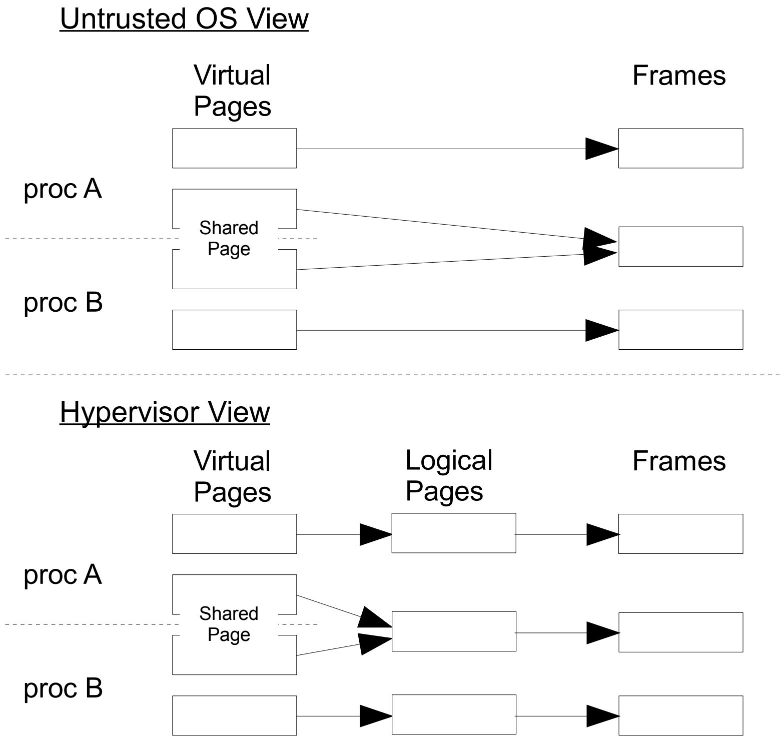


Figure 3: Virtual vs. Logical Pages

Invalid Sharing

Imagine that there exist two virtual pages that should not be shared, but the untrusted OS configures the page table such that they do. (See Figure 4 below.) How is corruption detected?

We'll assume, for this example, that the two pages originally share the same contents; perhaps they are COW copies of each other. Thus, the hypervisor allows them to share the same frame at first. (It is valid that they share the same frame, since they currently have the same contents. What is invalid, in this scenario, is that the OS makes one or both of the pages writable. When COW pages share a frame, all of the virtual pages must be readonly.)

When the protected process attempts to write through one of the virtual addresses, the hypervisor looks up the logical page for this virtual page. This logical page maps to a certain frame; this frame is currently shared with one other logical page.

Since this operation is a write, the hypervisor unmaps the **other** logical page from the frame. Thus, while there are two logical pages, only one is mapped to the frame; the hypervisor knows what the contents of the other ought to be (the old contents of the frame).

Now, the write proceeds, and modifies the frame. This virtual page (the one through which the write happens) has the correct data for now, since it points at the frame. However, its sibling page now has incorrect data, since the saved state shows that it should have the old contents, but the page table points it to the frame that has the new contents.

The hypervisor discovers this at some later point, when the protected process attempt to access data through the 2nd virtual page. The hypervisor attempts to map the 2nd logical page to the frame, but the contents do not match.

This is reported as a corruption, and the protected process is killed.

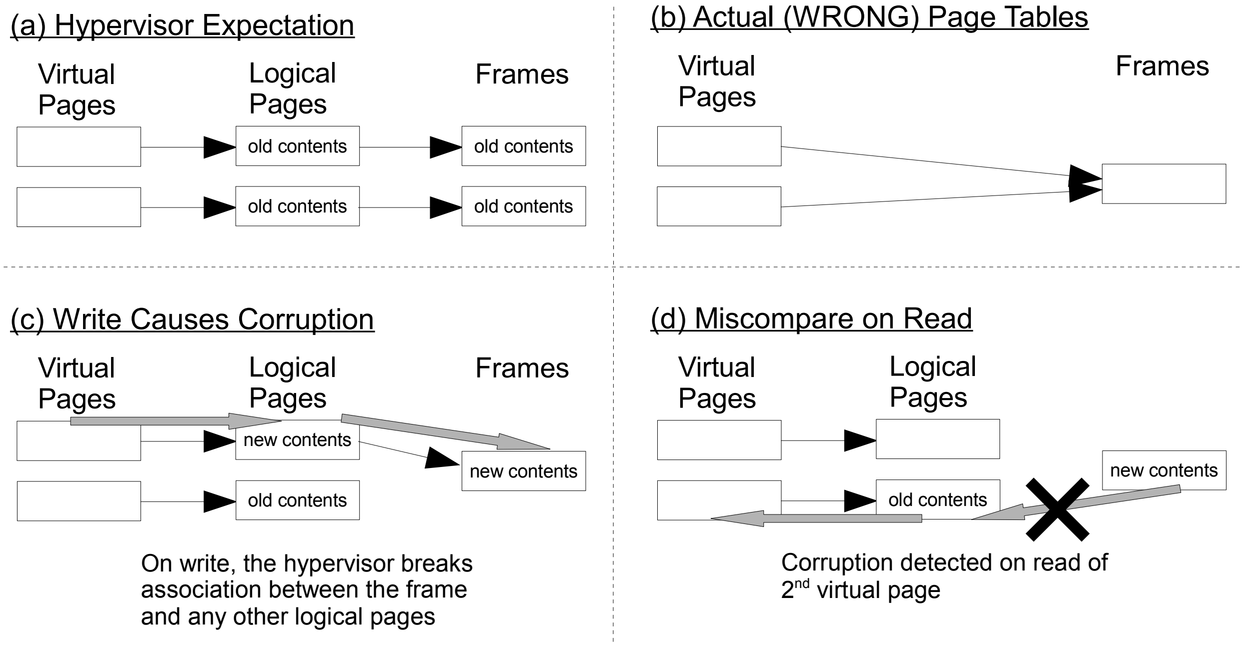


Figure 4: CORRUPTION EXAMPLE: Invalid Sharing

Failure to Share

Now imagine that there exist two virtual pages that should be shared, but the untrusted OS fails to do so. (See Figure 5). How is corruption detected?

In this case, the hypervisor knows that there is only one logical page, but the untrusted OS has set up page table entries pointing to two different frames.

When the write occurs, the hypervisor maps the virtual page to the logical page. There are no other logical pages mapped into that frame, so the write is allowed to proceed without problem. The write changes the contents that are stored in that frame.

As with the previous example, the virtual page through which the write occurred has the correct data, since it points at the frame that contains the recently modified data. However, its sibling now has the wrong data, as it points to the wrong frame.

The hypervisor detects the corruption when the protected process attempts to access the 2nd virtual page. The logical page is currently mapped into the 1st frame, but the page tables tell the hypervisor that this virtual address should map to the 2nd frame. Therefore, the hypervisor unmaps the logical page from the 1st frame, changing that frame to contain the signature for that data. It then attempts to map the logical page into the 2nd frame, but the contents do not match.

The hypervisor reports this as a corruption, and the protected process is killed.

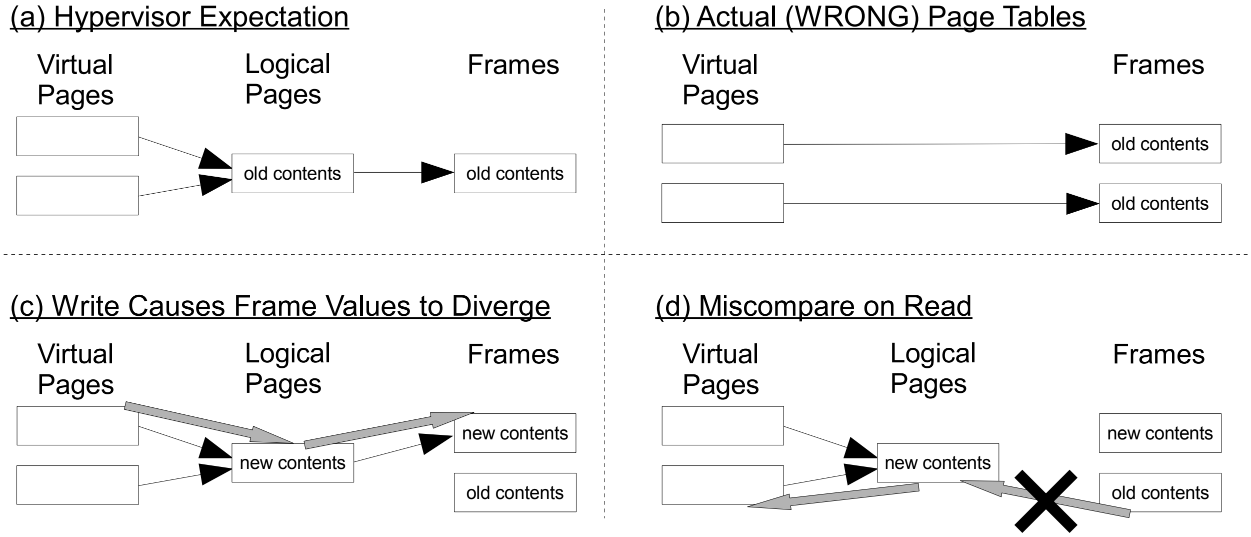


Figure 5: CORRUPTION EXAMPLE: Failure to Share

3.2 Snooping Attacks

Snooping attacks are where the attacker attempts to determine the private data of the protected process. These include direct attacks, where it attempts to directly read memory or registers, and indirect attacks, where it attempts to corrupt the state of the program such that it will expose its private data. Since we already have mechanisms to prevent corruption, this section will discuss only direct snooping attacks.

Private pages in memory are protected with signatures. As outlined in Chapter 1, Bodyguard ensures that the protected application always sees the private data, while untrusted code always sees the signature which represents the data. (See Figure 6 below for more details on how this is implemented.)

To protect registers, the hypervisor automatically wipes most registers when code jumps out of trusted code into untrusted code for any reason (branch, syscall, interrupt, exception, breakpoint, etc.). Thus, even if the operating system uses interrupts or breakpoints to intercept the application, it cannot read the private data stored in registers. (The contents of these registers are automatically restored when the CPU eventually returns to the protected process.)

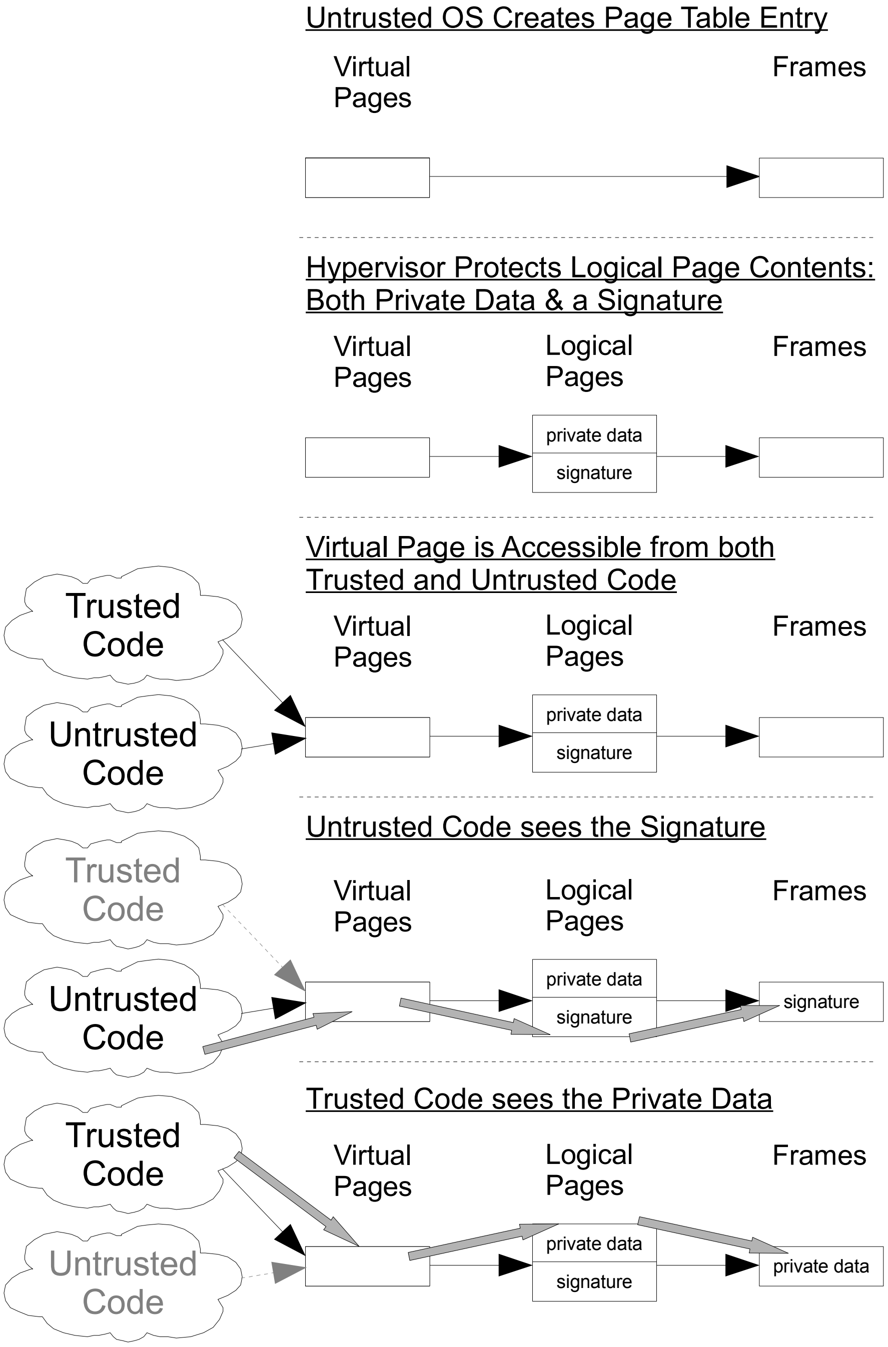


Figure 6: Signatures Prevent Snooping

IPC Snooping

An attacker may attempt to snoop on the data carried on IPC mechanisms between processes (for instance, reading the data on a pipe between ssh and the main process). As noted above, we expect the trusted application to either encrypt its IPC, or (as discussed in Future Work) emulate it using shared memory. In either case, snooping will be prevented.

I/O Snooping

An attacker may attempt to read the contents of a file or network stream used by the protected application. File snooping may be prevented simply by ensuring that all file I/O uses page-aligned private pages; when the OS tries to write the page to disk, it will write the signature. As detailed later, these pages can be later read up into the process, into new virtual pages, and the hypervisor will automatically recognize the signatures and show the private data to the process. (While this mechanism trivially prevents snooping, it doesn't prevent corruption. As discussed above, if corruption is a concern, the process will also need to use validate the contents of such pages.)

As discussed above, this thesis assumes that the trusted application uses ssh, with a protected IPC channel, to encrypt (and thus prevent snooping on) network traffic.

Simultaneous Attacks

Since memory snooping is prevented by replacing pages with signatures, and the private data is automatically restored when the protected application accesses it, an attacker may attempt (on an SMP system) to snoop by running an attacking process at the same time as a protected process, and accessing the private data through an inappropriate mapping to the frame.

The hypervisor handles this easily. When a frame contains private data, no untrusted code, on any processor, is allowed to access the page. If any untrusted code attempts to do so, the hypervisor will intercept the access, replace the page with its signature, and then allow the access to proceed.

Likewise, while the hypervisor contains the signature, no protected process, on any processor, is allowed to access the page. If any protected process attempts to do so, the hypervisor will intercept the access, restore the private data, and then allow the access to proceed.

Thus, if an attacker attempts simultaneous access, it may cause very poor performance (because the page is constantly switching back and forth between the signature and the private data), but the attacker will never be able to see any private data, nor cause the protected processes to see improper data.

DMA

An attacker may attempt to read or write private data pages by instructing a device (such as a hard disk or network card) to read or write physical pages using DMA [22]. Any such access, by any device, is treated as an access by untrusted code. DMA is thus simply a special form of a simultaneous access attack: the device will see signatures, while the protected process will see the private data.

Replay Attacks

In a replay attack, the untrusted operating system runs multiple copies of the same protected application, perhaps duplicating it and running several in parallel, or perhaps rewinding it to a previous state and running it again. The attacker's hope would be that, over time, it might be possible to find a workable attack (for instance, by feeding the application random inputs in hopes that some are not handled appropriately).

Bodyguard prevents this implicitly. There is only one copy of the process in the hypervisor's tables, and thus only one set of valid registers and one valid working set. If an attacker were to attempt to rewind the process to a previous state, even if that state is an exact duplicate of a previously-valid state, the hypervisor will view this as register corruption (and/or memory corruption) and crash the protected process.

Thus, it is impossible to even attempt a replay attack.

Dictionary Attacks

An attacker might attempt to expose the contents of a private data page through a dictionary attack. For instance, it might run its own protected process and read the pages at various points in time, hoping to map signatures to private data. Or, it may attempt to gain insight into the process, looking for duplicate pages, by scanning a process's working set for pages with duplicate signatures.

Dictionary attacks fail because signatures are randomly generated[3]. Thus, dictionary attacks are impossible, since two different pages, which happen to have the same private data, will have different contents when read by untrusted code[4].

3.3 Valid OS Page Accesses

This section discusses various **valid** ways in which the OS might access pages that could contain private data. We will also provide an overview of how the hypervisor should handle each scenario.

3.3.1 mmap()/munmap()

The program may ask the operating system to `mmap()` new pages, or `munmap()` existing pages. These actions change the working set of the process in valid ways. However, from the perspective of the hypervisor, sometimes these actions will look like corruption. For instance, imagine that a certain page was mapped into the protected process, and known to the hypervisor. The process unmaps this page, and maps another page in its place. Although this is a valid change, the hypervisor will perceive this as an attempt to corrupt the contents of that page.

Therefore, the hypervisor depends on the protected application to inform it when such operations occur. This allows the hypervisor to tell the difference between valid working set changes and corruption. This thesis performs most of this work implicitly and lazily.

3.3.2 write() and Similar Syscalls

Some syscalls, such as `write()`, legitimately read the private contents of a page. To accomplish this safely, a protected application performs two additional steps. First, it copies the data to a “bounce buffer” (initializing any unused space in the buffer, to prevent leak of old private data that might reside there). Second, it notifies the hypervisor that the page(s) of the bounce buffer are now “public” pages, meaning that they contain data that may be shown to the operating system. Once steps these are accomplished, the protected process actually performs the syscall; when the OS attempts to read the buffer, it is allowed to see the correct data because the protected process marked the page(s) public.

3.3.3 read() and Similar Syscalls

Some syscalls, such as `read()`, legitimately write to virtual pages. Such modifications should not be considered corruption by the hypervisor.

[3] In Overshadow (see Chapter 5), the pages are encrypted using a standard (but private) key. However, they still prevent dictionary attacks; they generate a random initialization vector for each new encryption action. Thus, two encrypted pages, encrypted at different times, will have different ciphertext values.

[4] The only exception is COW pages, where two pages might have the same signature because the signature for both was generated at the same time, when they were both mapped into the same frame. It is no leak, though, to tell the untrusted OS that these pages were duplicates of each other; it already knew that, and that is why they were both mapped to the same frame.

To accomplish this, the protected process must first allocate a new buffer in the process space (initializing it if necessary, to prevent a leak of old data that might have resided there before). Second, it tells the hypervisor to forget the contents of that buffer, effectively removing it from the working set of the protected process. Third, it performs the syscall. Fourth, the protected process accesses the virtual pages, which implicitly brings them back into the working set. Finally, it copies the data from this buffer into the destination buffer in private space.

3.3.4 COW

In some situations, the operating system may perform COW on private pages. For instance, the protected process might `fork()` itself; then there exist two virtual pages for each physical page, and as the processes run, the OS must perform COW on the various pages. That is, the operating system will (upon the first attempt to write to the COW page) read the physical page, copy it to another, and then update the page tables such that the virtual pages no longer share the same physical page.

Fork

To handle this, the hypervisor duplicates the virtual and logical pages of the process, at the moment when the `fork()` occurs. It initializes their contents to be duplicates of each other (namely, it records that 2 logical pages now map to the same physical), but the new logical pages do not share any logical link. Later, when the OS needs to copy the page, the hypervisor will generate a single signature, which is saved as the "current contents" of both logical pages, because both are mapped into the same frame. When these pages are accessed again (by the protected processes), the hypervisor restores each from the signature, entirely independent of the other.

Other Instances of COW

COW can also happen when the untrusted OS knows that two pages map to the same physical page in their initial state. For instance, when the process is bootstrapped, the same physical page may be mapped into one location as a (readonly) code page, and mapped into another location as a (read-write) data page. The untrusted OS will point both of these virtual pages to the same physical page until the data page is modified.

The hypervisor allows the initial state of the protected process to include a plurality of pages that all have the same private contents, and the same signature. The untrusted OS selects one frame, and fills it with the signature; any number of logical pages may be mapped to the same frame.

3.3.5 Swap

At any time, an operating system may choose to swap a page to disk. This, of course, means that the untrusted OS (or, DMA) will read the contents of a private page, and later write it back.

In this scenario, the untrusted OS sees the signature, rather than the private data. Thus, what is written to disk is the signature. Later, when the protected process attempts to access the page, a page fault will result, and the OS will restore the signature. Possibly, it will reside in a different physical page than before. However, since the hypervisor only cares about the **contents** of pages, not their physical implementation, it will confirm that the logical page has the correct contents, swap the signature back to the private data, and allow the process to use its memory.

3.3.6 Signals

If the process registers for signals, then the untrusted OS is justified in sending signals to the process at any time (except when they are masked), forcing code into the defined signal handlers. Note that this will look like register corruption, since the kernel will change the IP (and perhaps the SP). It may also look like virtual space corruption, if the kernel has to write out values to the virtual space, such as creating a stack frame.

This paper does not describe how signals might be handled. We hope to implement a secure and general solution in our Future Work.

3.4 Summary

In this chapter, we detailed the various types of attacks that are possible. We discussed valid operations that an uncorrupted OS might perform. It also sketched Bodyguard's protection mechanism and how it applies to these various operations.

Chapter 4: Bodyguard Design

This chapter starts by describing the hypervisor. It then describes an “entity,” which is our protection boundary.

Next, it details the properties of a protected process, including discussing how the hypervisor knows when one is running. It describes the characteristics of the virtual space of a protected process, discussing the working set, virtual pages, and logical pages. It discusses how the hypervisor handle physical frames, and how it controls access to those frames.

Next, the chapter will discuss more abstract concepts, such as signatures, how the hypervisor maps the guest page table to the effective host page table, and how it protects registers.

Next, the chapter will detail how the shim works (including the syscall handler). The shim is a mechanism we use to adapt a legacy process to Bodyguard protection without having to rebuild it.

Next, the chapter will discuss how the client and hypervisor work together to initialize a new entity in a secure manner.

Finally, the chapter will present a number of detailed examples, demonstrating how Bodyguard provides protection in a practical environment.

4.1 Hypervisor

The *hypervisor* is a hardware or software component that implements Bodyguard's basic protections. In our prototype implementation, the hypervisor is special code running inside the Bochs x86 emulator. Thus, our experimental setup effectively emulates a new hardware implementation. However, we believe that the ideal setup (to be explored in Future Work) would be a VMM, such as Xen [23].

The hypervisor supports a set of *hypercalls*, which are direct calls from the protected application to the hypervisor. The hypervisor interprets the call, and determines the return value; Bochs then swallows the operation (effectively turning it into a NOP) so that the untrusted OS never sees it.

Ideally, Bodyguard would use a new instruction for this, but our work uses a simpler approach (since it is currently limited to Linux). The hypervisor inside Bochs monitors all system calls. If any application uses system call 400, which is undefined in Linux, it is interpreted as a hypercall.

In most cases, hypercalls and syscalls are separate operations. However, there are a few cases where it is necessary to perform a hypercall, and then atomically perform some syscall (without ever returning to the protected application). Currently, these few operations are hard-coded for the Linux system call numbers and system call mechanism, but this could be generalized in the future.

4.2 Entities

In Bodyguard, the hypervisor keeps track of one or more *entities*. An entity is a single instance of a protected application, encompassing one or more processes. All of the processes in the same entity have the ability to read and write each other's private data pages (if the appropriate mappings are set up, of course), but nothing outside of the entity (including untrusted code mapped into any of these processes, such as kernel code) will be allowed to read or write any of the private pages. Several entities can be running in the same untrusted OS at any given time.

We describe the entity initialization process later in this chapter.

4.2.1 Destroying an Entity

The hypervisor destroys an entity in three different circumstances:

1) When the last process within it dies. (See below for a description of process destruction.)
2) When the entity fails any sort of check, and the hypervisor believes that the state has been corrupted.
3) When the link between the hypervisor and the client drops. (See below for a description of the link to the client, which is also used for initialization.)

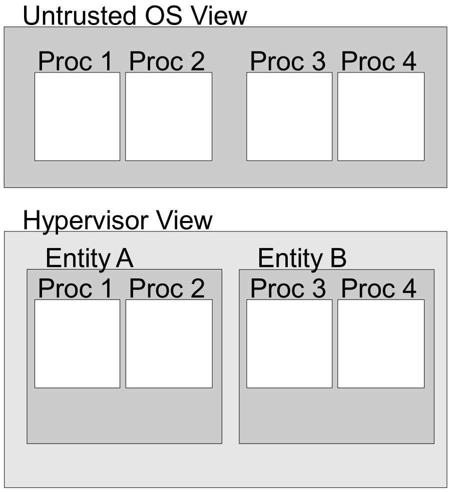


Figure 7: Entities and Processes in the Untrusted OS

When an entity is destroyed, the hypervisor deletes each process within it. This involves, of course, deleting all of the virtual and logical pages within each. If any of the logical pages are currently mapped into any frame (showing the private data, not a signature), the hypervisor ensures the confidentiality of the private data by replacing the data with a signature before the logical page is destroyed. (See below for details about how a logical page is destroyed.)

Once all of the processes, virtual pages, and logical pages have been destroyed, the hypervisor destroys any record of any saved pages or signatures associated with this entity.[5]

Finally, the hypervisor reports status to the client (if the link is still up), and closes the link to the client.

In some cases, the untrusted OS may continue to run a protected process after its entity has been destroyed. (This can happen if the client link dies, or if the hypervisor detects corruption.) The process will die very quickly in this state, because the code pages are all converted to signatures (with no way to restore them). Thus, as soon as the process runs its next instruction, it will execute random data, and the process will immediately fail with an illegal instruction exception.

Lingering Entities

As mentioned below, it is possible for a protected process to die without notifying the hypervisor. This means that it is possible for an entity to linger after the last process has died (because the hypervisor still thinks that one or more processes are running). A lingering entity may tie up hypervisor memory, since the hypervisor must keep around copies of all of the signatures ever created for this entity. (See Section 4.7: Signatures and Saved Pages below for a discussion why.) However, a dead entity should get cleaned up relatively quickly, as the client should become aware of the situation before long, either by receiving a completion message through the network, losing the network connection, or simply timing out. When the client detects the termination, it will drop its connection to the hypervisor, and the entity will be cleaned up automatically.

[5] As detailed below, the hypervisor is required to keep track of every signature ever created, and the private data that each maps to. This data can be discarded when the entity dies, since there will never again be a need to recognize a signature and restore the private data.

4.3 Processes

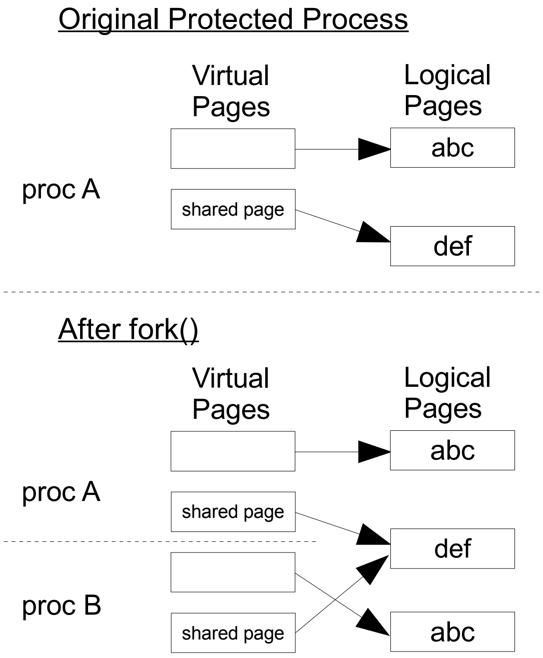
A *protected process* is a process within the untrusted OS that is known to the hypervisor and receives protection for its memory and registers. Each entity includes at least one protected process.

4.3.1 The First Process

The first process in an entity is created using the initialization method described later in this chapter. The initial state of the working set, and the initial state of the saved registers, are determined by the client and communicated to the hypervisor.

4.3.2 fork()

All protected processes, other than the first in each entity, are created with `fork()`[6], duplicating an existing protected process. When the process performs a `fork()` within the untrusted OS, the hypervisor is also notified and duplicates the process.



Duplicating the process requires that the hypervisor duplicate the working set of the process. (See Figure 8.) Each virtual page is duplicated in the new process. Virtual pages that point to shared logical pages will point to that same logical page in the new process. However, virtual pages that point to non-shared logical pages will point to new logical pages, which are duplicates of the ones from the existing process. The result is that pages that must be shared will share a common logical page, and pages that need to not be shared will not share logical pages. Thus, the hypervisor can enforce that the untrusted OS properly implements COW semantics.

Likewise, the saved register state is copied to the new process.

Atomicity

`fork()` has some unusual complexities because both the hypervisor and the untrusted OS need to duplicate the process state, and they must duplicate precisely the same state.

A naïve implementation of `fork()` might have the protected process first hypercall to the hypervisor, asking it to duplicate the process state, then return to the process and perform the `fork()` syscall. However, this will not work, because the process state might change between the hypercall and the syscall. Thus, when the child process eventually ran, the hypervisor would believe that corruption had occurred.

Instead, Bodyguard implements a hypercall that atomically performs the hypervisor copy, and then calls the syscall on behalf of the process, without ever returning to the process. Thus, the process state that the hypervisor copies is exactly the same as the process state that the untrusted OS copies. Thus, when the child process runs, it will have exactly the state that the hypervisor expects.

[6] The current version of this thesis was designed for use with Linux, which uses `fork()` to create new processes. In the Future Work, we may investigate how to generalize the design to handle Windows and other operating systems.

4.3.3 Lingering Processes

The hypervisor does not have any automatic way to know when a process dies in the untrusted OS[7]. Instead, the hypervisor expects a dying protected process to use the "kill-process-then-call-exit" hypercall. Like the fork hypercall, this is an atomic hypercall; it first cleans up the current protected process inside hypervisor memory, and then immediately calls Linux's `exit()` syscall.

If a protected process fails to die cleanly (say, because of `kill -9`), then the hypervisor will not realize that this process is dead, because this hypercall will never run. However, as we noted above, the entity will die eventually, for some reason; at that time, the process will get automatically cleaned up. A process that thus lingers in the hypervisor memory is harmless, other than that it consumes a small amount of memory for its working set table.

4.3.4 Jumping Into a Process

Nothing ever explicitly informs the hypervisor when the untrusted OS schedules a protected process to run. Instead, the hypervisor implicitly discovers that a protected process is running when it discovers that the CPU is executing code that resides in a private page. [24] Since each private page is owned by exactly one entity, the hypervisor knows which entity is running, based solely on the private page. However, the entity may include multiple processes, and so the hypervisor needs a way to tell which process is running. It does this by inspecting the process ID[8].

Once the process ID is known, the hypervisor looks up the saved register state for this process. It confirms that the current contents of the registers are correct, and then restores the contents of any registers that were wiped when the process last left protected code.

Once the registers are restored, the hypervisor looks up the instruction pointer, and checks the contents of that page. That is, it looks up the correct virtual page in the working set of this process, compares the saved contents[9] with the frame.

If the contents of the code page are correct, then the protected process is allowed to run. However, if any of the above checks fail, the hypervisor interprets this as corruption and destroys the entity.

Running Untrusted Code Inside the Process

Kernels typically map their code and data pages into the same context as user processes. Thus, the hypervisor will often find that a process with a known process ID is executing untrusted code pages. This is allowed, but the code is **not** said to be executing the protected process; it is executing untrusted code. Therefore, the code has no right to read or modify private pages.

[7] We could implement a handler for SIGCHLD to catch processes as they die, but we would still have the fundamental problem that there would be some "grandparent" process of everything, which could be killed without the untrusted OS knowing about it. Thus, we have to consider the possibility of lingering processes, even if we eventually implement a SIGCHLD handler to make them less common.

[8] In our current design, we use the page table pointer as the process ID, as we have never observed it to change in our system, during the lifetime of a process. However, we realize that this is not necessarily reliable, and a better method should be found.

[9] The hypervisor always requires that the protected process bring all code pages into the working set (with reads or writes) before attempting to execute them. If the process attempted to execute a virtual page which was not yet in the working set, then an attacker could drop in any arbitrary page, and thus corrupt the process. Thus, we can always assume that the page to be executed is in the working set of the current process.

4.3.5 Jumping Out of a Process

Likewise, nothing ever explicitly informs the hypervisor when untrusted code is running. Instead, the hypervisor implicitly discovers this when it discovers that the CPU is executing code that resides in a public page.

Any jump from protected code into unprotected code, for any reason (fall-through, call or branch, interrupt, exception, or breakpoint) is treated the same way; the hypervisor notes that the processor is now executing untrusted code. Any attempt to access private pages will be intercepted, as detailed below.

4.3.6 Example: Syscall

Imagine that a protected process is running, and it performs a syscall. Before it executes the syscall, its instruction pointer points to a private page. Since this is a private page, and the process ID is recognized, we say that Bodyguard is running a protected process. This means that the code may both read and write private pages associated with this entity.

When the process performs a syscall, this forces the code to jump to a different page, which is **not** a private page (it is kernel code). Since the process is jumping out of private pages into public pages, the hypervisor (as detailed below) saves the registers and zeroes out most of them (leaving only the critical ones necessary for the syscall). It also disables any access to private pages. The process executes code within the kernel.

When the kernel is ready to return to the protected process, it performs the return-from-exception instruction. Amongst other things, this instruction jumps the process back to the instruction after the syscall. This, of course, is back in private code pages. The hypervisor confirms that the instruction pointer and registers are correct (that is, that they match the saved values). It also verifies the contents of the code page that is being executed. If all checks out, then the protected process starts executing again.

It's important to note that all of this happened within the context of a single process. The process ID never changed; the guest OS page tables may not have changed. However, the hypervisor automatically detected the transition from the protected process to the kernel (and back again), and was able to confirm that the kernel did not corrupt the state of the process. All of this happens implicitly, without any explicit action by the process. The hypervisor can detect these transitions simply by checking which code pages are public and which are not.

4.3.7 Threads

We have not implemented support for multiple threads in a single process, but it should be fairly easy to generalize the register-snapshotting mechanism (see below) to support multiple buffers (one buffer per thread). One key difficulty is that we must find some way to determine which thread is running when code jumps into private pages. Some architectures already store the thread ID in a register, which would solve the problem; another option is to treat the instruction pointer/stack pointer pair as a unique identifier. We leave this as something to examine in the Future Work.

4.4 Working Set

The *working set* of a protected process is the list of virtual pages that are in this process's virtual address space and that are known to the hypervisor.

Adding a New Page

New pages are added to the working set lazily, when they are first read or written by the protected process[10]. The initial contents of the page may be a signature, or they may be arbitrary data of any other kind. If it is a signature[11], then the page is initialized as a private page; before the protected process is allowed to perform the access, the hypervisor replaces the signature with the private data associated with that signature.

On the other hand, if the physical page does not contain a signature, then the virtual page is initialized as a public page (just as if the process had written this data to a page and then told the hypervisor to "mark it public"). Since the page is public, untrusted code will be able to read its contents until the application modifies it. Once the application modifies it, it becomes a private page.

This implicit design elegantly handles new anonymous pages (mapped with MAP_ANONYMOUS), which are normally zero pages. It also handles `mmap()` of files where the contents of the file are public data. Moreover, it handles situations `mmap()` of files where the contents of the file are signatures (written at some earlier time by this same entity).

However, a downside of this design is that the protected process must verify the contents of each new `mmap()`, since the hypervisor has no idea what to expect. For instance, with a new page mapped with MAP_ANONYMOUS, the page may not actually be initialized to zero as it was supposed to be; an `mmap()` of a file may not reflect the true file contents.

Removing Pages from the Working Set

Any time that the process performs `munmap()`, it must remove the page(s) from the working set. This is so that later, if the process `mmap()`s new pages to the same virtual addresses, the hypervisor will not interpret this as corruption. It accomplishes this with a "forget range from working set" hypercall.

Likewise, any time that the process is about to perform any syscall that will modify any buffers in the virtual space, it must remove the page(s) from the working set before the syscall. Again, this is so that changes to the pages will not be interpreted as corruption by the hypervisor.

Destruction of the Working Set

When the process dies, all of the virtual pages in the working set are automatically cleaned up. Of course, if some of those pages point to shared logical pages, then those logical pages will persist; however, most of the virtual pages will point to non-shared logical pages, and those logical pages will get cleaned up as part of process destruction as well. (Later, we will discuss exactly how Bodyguard cleans up logical pages.)

[10] The hypervisor never imports a page into the working set based on an attempt to execute a page, which means that a protected process cannot simply jump into a newly `mmap()`ed code page. If a protected process attempted this, an attacker could drop in any contents that he desired into that page. Even if the hypervisor knows that the code page is a private page (that is, the frame contents match a signature), it doesn't know that the attacker mapped in the **right** code page. Instead, the protected process must always read a code page, and confirm its contents, before its first attempt to execute it.

[11] It must be a signature of a private page associated with this entity. If one entity attempts to access the private data of another, this is treated as a normal access by untrusted code, and the attacker will see the signature, not the private data.

4.5 Virtual and Logical Pages

As discussed in Section 3.1.3: Shared Memory Attacks above, Bodyguard distinguishes between virtual and logical pages. A *logical page* represents the logical contents of a page; a *virtual page* represents one page in the working set of a protected process. There is a many-to-one relationship between virtual and logical pages; if multiple virtual pages map to the same logical page, then this means that all of these virtual pages are shared pages, all the protected process expects all of them to be mapped to the same physical backing page.

Initialization

When a protected process reads or writes a virtual page which is not in its working set, Bodyguard creates new virtual and logical pages to represent it. The new virtual page points to the new logical page, and the virtual page is added to the working set of the process.

If the physical page (as determined by the page table entry) contains a known signature from this entity, then the logical page will be initialized as a private page. Its current contents will be equal the private data associated with this signature.

If the physical page does not contain a signature[12], then the page will be initialized as a public page. The current contents will be the current contents of the physical page. Like any public page, the page will become private if/when the protected process writes to it.

4.5.1 Public vs. Private

Most logical pages are private. *Private* means that the page contains (or might contain) private data that the process does not want a snooping attacker to be able to read. If an attacker attempts to read a private page, it will see a signature instead.

Public pages are those that the protected process has declared are safe to share with untrusted code, or recently added pages whose contents were initialized by the kernel. They often contain data that must be sent to the kernel, but may also include recently allocated pages that have not yet been modified.

If the protected process writes to a public page, it automatically transitions from public to private. However, the contents of the page (as seen by the protected process) will still mostly contain the old data from the public page (until overwritten).

Both types of pages are protected from modification. That is, the hypervisor keeps track of the current contents of both types of pages, and untrusted code is not allowed to modify it.

4.5.2 Shared Pages

A *shared logical page* is a logical page that represents a shared page explicitly declared by the protected process. It represents a shared page mapping that may be shared amongst various processes in this entity.[13]

A process declares an existing[14] virtual page to be shared with a hypercall. The hypercall gives the

[12] ...or contains a signature from some other entity!

[13] Note that “shared” means “shared between processes in this entity,” not “shared with untrusted code.” Shared pages are still private pages, unless you explicitly make them public. (The author cannot imagine why this would be desirable, but it is conceivable.)

[14] We do not allow the process to declare a page shared until it is in the working set.

virtual address and the shared page identifier. The hypervisor searches through the entity's list of shared pages, looking for a logical page with that identifier. If it finds one, then it associates the virtual page with that preexisting logical page[15]. However, if it does not find one, then it it creates a new shared logical page, associates the virtual page with that, and adds it to the entity's list of shared pages. The old logical page is destroyed.

Thus, if two processes (or the same process, at two different times), declares two shared virtual pages with the same identifier, then those virtual pages will point to the same logical page until they are removed from the working set.

This means that the various processes in an entity must have a shared table in virtual memory (see Figure 9: The Shared Page Table and Figure 10: Allocating a Shared Page) that is used to assign and store identifiers. Since we cannot trust the untrusted OS to handle communication between the processes, the processes will use shared memory to implement this table. Each process, as part of its initialization, uses mechanisms in the untrusted OS (shmget() or similar) to allocate a shared virtual page. It then notifies the hypervisor about this page, using a magic value[16] as the identifier. This page becomes the root of a data structure of shared pages through which the various processes may maintain a table of allocated identifiers; each entry in the table maps an identifier in the untrusted OS world to an identifier in the table.

shmget() identifier	hypervisor identifier
0x1234	10
0x5678	11

Figure 9: The Shared Page Table

```
key = shmget(...);
private_key = lookup(key);
addr = shmat(key, ...);
...touch addr...
hypercall(..., private_key, addr)
```

Figure 10: Allocating a Shared Page

For instance, when a shared memory region is declared in the untrusted OS, the page(s) of that region are assigned private identifiers, and the association is stored in the table. All protected processes, which map to the shared memory, will thus give the same ID to the hypervisor. This allows the hypervisor to understand which virtual pages in the various processes are shared logical pages.

There is no way to turn a shared logical page into a non-shared one. However, it is possible to delete a virtual page that points to a shared logical page, and then to touch the virtual pages to bring them back into the working set as non-shared. (Note that this means that there is a window of time where the virtual page will not be in the working set, and thus an attacker could change the contents without detection. If you care about this, then copy the contents to another page, break the link, and then confirm that the contents didn't change.)

4.5.3 Initialization

[15] The hypervisor will demand that the two logical pages – the old shared one, and the new one which is just being declared shared – have the exact same contents. This will, of course, be true if shared memory is working correctly, since the two logical pages should already share the same frame. If the pages have different contents, Bodyguard interprets this as corruption, and kills the entity.

[16] The magic value will be hard-coded into the design of the trusted application (or the shim, if that implements this shared table); the only requirement is that all processes know to use this same value for the bootstrap of their copy of the table. Note that it is fine if the attacker knows the magic value; it is not a secret, simply a convention used for bootstrap. The attacker, even if he knows the magic value, cannot access the page, because any attempt to map the shared page from untrusted code will be rejected. (Untrusted code is not allowed to modify the working set of any protected process.)

4.5.4 Saved Contents

The hypervisor always keeps track of the contents of each logical page. Sometimes, this is accomplished implicitly, by recording that a certain physical frame contains the data. Other times, this is accomplished explicitly, copying the contents to private hypervisor memory. If this is a private page, the hypervisor will also keep the signature in hypervisor private memory.

4.5.5 Destruction

Any time that the hypervisor destroys any logical page for any reason, it checks to see if that logical page currently resides in any frame. If so, it unmaps the logical page from the frame. If this was a private logical page, and if it was the last logical page mapped into the frame, the hypervisor will drop the page's signature into the frame before unmapping it. This will ensure that no private data is forgotten and left around in old frames. However, a frame may continue to hold the private data if there exist one or more surviving logical pages still mapped into it.

4.5.6 State Machine

Sometimes, a logical page is mapped to a frame. Sometimes, it is not. Sometimes, the frame contains a signature; sometimes, it contains private data. We can model this with a state machine (see Figure 11).

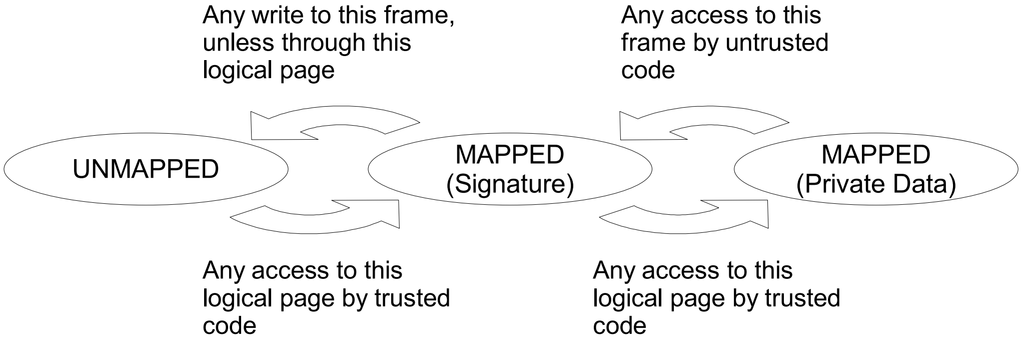

Any write to this frame, unless through this logical page
Any access to this frame by untrusted code
UNMAPPED
MAPPED (Signature)
MAPPED (Private Data)
Any access to this logical page by trusted code
Any access to this logical page by trusted code

Figure 11: Logical Page State Machine (Private Pages)

In this state machine, transitions are always driven by accesses, either by untrusted code or trusted code. Accesses by untrusted code drive the state towards the left of our figure; accesses through this logical page by protected processes drive the state towards the right.

The three possible states are:

- UNMAPPED: The page is not currently mapped into any frame.
- MAPPED (Signature): The page is mapped into a specific frame, and the frame contains the signature for this logical page.
- MAPPED (Private Data): The page is mapped into a specific frame, and the frame contains the private data for this logical page.

There are only four transitions possible:

- UNMAPPED → MAPPED (Signature): Confirm that the contents of the frame match the signature saved for this logical page. If they match, then it is OK to map the page to the frame. Otherwise, the hypervisor detects corruption.
- MAPPED (Signature) → MAPPED (Private Data): Replace the signature with the private data, so that protected code can access the data.
- MAPPED (Private Data) → MAPPED (Signature): Generate a signature (if one doesn't already exist), and replace the private data with the signature.
- MAPPED (Signature) → UNMAPPED: Break the association between the logical page and the frame. Record the expected contents of the logical page as you unmap.

Public Pages

Public pages have the same basic state machine, the difference being that they do not have signatures, and thus do not need a MAPPED (Signature) state. (See Figure 12.)

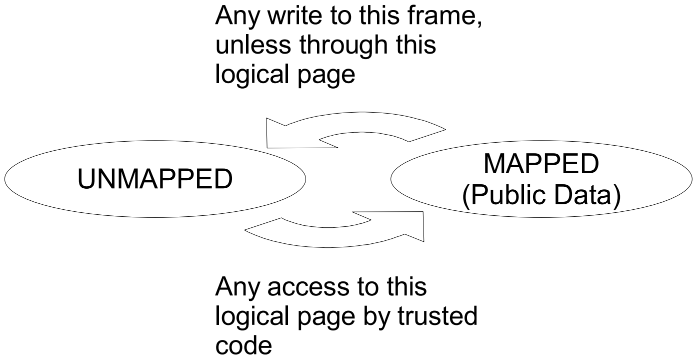

Any write to this frame, unless through this logical page
UNMAPPED
MAPPED (Public Data)
Any access to this logical page by trusted code

Figure 12: Logical Page State Machine (Public Pages)

4.6 Physical Pages (a.k.a. Frames)

A *physical page (a.k.a. frame)* is a guest-physical page; that is, it is a physical container for one page worth of data.

Frames are managed by the untrusted OS. It decides which pages from which processes should reside in which frames. It performs COW; it performs swap; it initializes new processes, and cleans up old pages after a process dies.

The job of the hypervisor is to monitor the actions taken by the untrusted OS and confirm that they do not allow corruption of, or snooping on, protected processes. To accomplish this, the hypervisor keeps track of which logical pages (if any) are mapped into each frame at any given time.

A frame may be modeled as a state machine with four states (Figure 13)[17].

- NO_MAPPING: No logical page is mapped into the frame
- PUBLIC: One or more public logical pages are mapped into the frame

[17] Note that this state machine is effectively the merging of the private logical page and public logical page state machines given above.

- SIGNATURE: One or more private logical pages are mapped into the frame; the frame currently contains the signature for those pages.
- PRIVATE: One or more private logical pages are mapped into the frame; the frame currently contains the private data for these pages.

It is worthwhile to look at the frame state machine, in addition to the logical page state machines, in order to understand what happens when untrusted code attempts to access the frame. First, untrusted code is never allowed to access the frame while it is in the PRIVATE state; if it attempts any such access, the frame will be driven to the SIGNATURE state immediately. (Of course, this means that the private logical pages are each driven to the MAPPED (Signature) state as well.)

Read accesses by untrusted code are permitted while the frame is in the NO_MAPPING, PUBLIC, and SIGNATURE states. However, write accesses by untrusted code are only allowed in the NO_MAPPING state (so that the write doesn't alter the contents of a logical page). Thus, if untrusted code attempts to write to a frame, and one or more logical pages are mapped into the frame, those logical pages are forcibly unmapped before the write is allowed to take place.

Taken together, the logical page state machines, and the frame state machine, give a thorough understanding of how logical pages are mapped and unmapped. The logical page state machines are most useful when we want to understand accesses by the protected process; the frame state machine is most useful when we want to understand accesses by untrusted code.

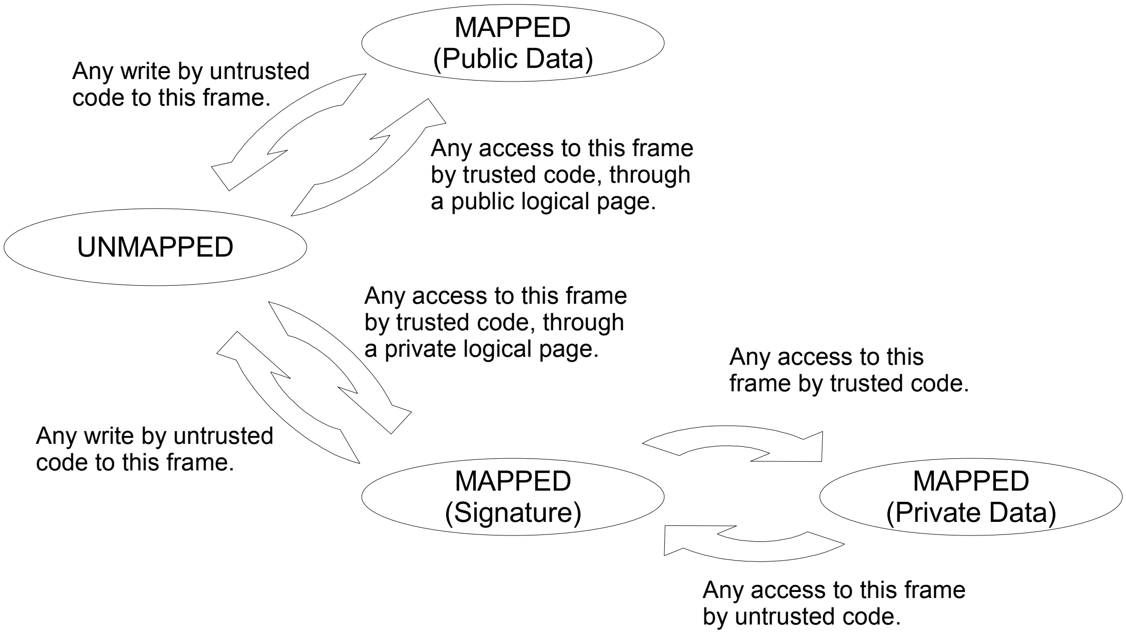


Figure 13: Frame State Machine (Private and Public Pages)

Lazy Updates

When the untrusted OS decides to unmap a certain virtual page from a certain frame (or map a new one in), it updates the page tables immediately. However, the hypervisor does not immediately notice the change. Instead, the logical page → frame mappings are updated lazily, the next time that the logical page or frame is accessed.

For example, imagine that a certain private logical page is mapped into a certain frame, with the private data showing. The untrusted OS decides to move this virtual page to swap. It therefore reads the frame contents. This read forces the frame into the SIGNATURE state; the OS then reads the signature, and writes it to disk. It then updates the page tables to indicate that this virtual page is no longer valid. The hypervisor, however, is unaware of this; since the frame has not been modified, the hypervisor still thinks that the logical page is mapped into the frame.

At some later point, the untrusted OS decides to map in a different page to the frame. It therefore writes the correct contents into the frame. This write is intercepted, since the hypervisor still has a logical page mapped in. The hypervisor unmaps the logical page from the frame before the write is allowed to proceed.

Thus, the hypervisor does not become aware of page mapping changes until there happens to occur a write to the frame, at some point **after** the untrusted OS actually unmapped the page. This is perfectly safe, because the hypervisor knows that the contents of the page didn't change during that window of time. Therefore, until the write actually is allowed to proceed, the frame still contains the signature of the logical page.

Likewise, when the untrusted OS maps in a new page to a frame, the hypervisor is unaware of the change until a protected process actually accesses the page. Again, there is a window of time where the page is mapped into the frame (from the untrusted OS's perspective), but the hypervisor doesn't know; this is safe because the hypervisor will perform content validation before the first actual use of the page.

Finally, if an attacker corrupts a logical page (that is, the untrusted OS allows a mapping to a frame with incorrect contents), the hypervisor also will not detect this until the first access. Again, the hypervisor doesn't attempt to map the logical page to the frame until the logical page is accessed by a protected process; at that moment, it performs the necessary content validation, and will detect the corruption. This means that if a page is never accessed after it is corrupted (or, if the corruption is fixed somehow, before it is accessed), then the hypervisor will not detect corruption. This is intentional; there is no need to crash the application if the corruption never actually affects the state of the protected process.

4.7 Signatures and Saved Pages

A *signature* is a page-sized block of data that uniquely represents the contents of a logical page at a point in time (or, a group of at-that-time identical logical pages). Any time that untrusted code attempts to access[18] a frame that contains private data, the hypervisor automatically generates a signature for that data (or, re-uses an old one if the page has not been modified since the old signature was generated) and replaces the private data in the frame with the signature. Thus, untrusted code, when it attempts to access any frame that contains private data, instead sees the signature.

A signature is made up of a magic string, followed by random data. Since the data is random, it is impossible for an attacker to read a signature and guess what the private data might have been.

Later, when the protected process attempts to access the page for any reason (read, write, or execute),

[18] It's obvious that a signature is necessary when untrusted code wants to read. But it is also true that a signature is required when untrusted code wants to write. Otherwise, an attacker could overwrite one byte and then read back the page and see the rest of the private data. Note that we don't have to worry about untrusted code trying to execute a private page. If the code jumps to a private page, this is interpreted as a jump into the protected process. This will result in either running the protected process, or detecting corruption and crashing.

the hypervisor automatically restores the private data. The hypervisor automatically bounces the various frames back and forth between the two states (signature and private data) as often as the access patterns require it.

4.7.1 Signature Persistence

Signatures may be used to initialize the contents of a new page. That is, the untrusted OS may take a signature that was saved at some point in the past, copy it into a frame, and then map a new virtual address to that page. This might be useful, for instance, if the protected process wrote out a page to a file (without using a bounce buffer); the file would contain the signature. Later, if the process mmap()ed that file to a new address, the hypervisor would detect that the frame contained an old signature; when the process accessed the page, it would see the old saved data. Alternatively, a programmer might implement a pipe between two protected processes, making sure to only copy page-aligned, page-multiple buffers; the receiving process would get the private data because the OS copied the signatures around. Of course, in both cases, the programmer must worry about validating the pages, but this might be accomplished with something as simple as a magic value and sequence number buried inside the private data.

There is no limit on how long the untrusted OS might hold an old signature before using it again. For this reason, a signature, once created, will be stored for the entire lifetime of the entity. This means that the total amount of hypervisor memory consumed by the entity will grow over time and without bound; however, all of this memory will be released the moment that the entity is destroyed. (We have considered alternative schemes where we might track which signatures are in use, and destroy them when they are no longer needed, but these are not part of the current design. They may be investigated in Future Work.)

4.7.2 Signatures and COW Pages

Sometimes, multiple private logical pages will be associated with the same frame. This happens when there are multiple logical pages that the untrusted OS knows to be COW copies of each other; it will map them to the same frame[19]. As the protected process(es) access each page in turn, the hypervisor will discover that they all reside in the same frame, and it will thus map the various logical pages to the same frame.

If untrusted code attempts to access the frame, the hypervisor generates a single signature and stores it in the frame. Note that this changes the state of all of the logical pages currently mapped into the frame; all of them move to the MAPPED (Signature) state at the same time (see Figure 11 above).

If it becomes necessary to unmap one or all of these pages from the frame, each logical page will point to the same signature as its “saved contents.” That is, even though we have multiple logical pages, we only have a single signature shared amongst them, because they were all mapped into the same frame. Of course, as the protected process writes to some of these pages, their contents may diverge; however, they will share a signature until that point.

[19] The untrusted OS can know that several pages are COW copies of each other without knowing their contents. One example of this comes about when we `fork()` a protected process.

4.7.3 Signatures and Readonly Pages

When a protected process attempts to read or execute a private page that previously was a signature, the signature will be retained. If it becomes necessary to switch back to a signature before the page is modified, the hypervisor will simply restore the old signature, rather than generate a new one. Not only is this convenient and fast, it is also necessary for correctness.

Remember that a signature is generated when untrusted code attempts to access a private page. The untrusted code may legitimately have copied that signature to another location, or even written it to disk (such as, but not limited to, swap). Now imagine that the hypervisor restores the private data and the protected process uses it for a bit. But then, before the protected process changes the page, the OS decides to discard this frame (swapping it out to make space for something else). The OS knows that the page is clean (that is, has not been modified since it was last copied or written to disk) and thus it is entitled to simply discard the frame without copying it again.

Shortly thereafter, the frame will be reused for some other purpose; when this access first happens, the hypervisor will put the signature back in place, and then a moment later record that the logical page is no longer associated with that frame (since untrusted code has written to the frame). If a new signature were created, then the saved contents of the logical page would reflect the new signature, but the saved copy that the OS has would reflect the old.

Thus, when the hypervisor needs a signature for a readonly page, it always chooses to simply re-use the old signature instead.

4.8 Masking Page Table Entries

In order to facilitate rapid execution of code inside a virtual machine, most VMMs translate the “guest page tables” (page tables defined by the guest OS) into an “effective page table.” This is some sort of data structure or set of mappings which allows code inside a virtual machine to access its virtual pages without the VMM having to intercept and manually translate each access.

Throughout this document, we have said that the Bodyguard hypervisor will intercept memory accesses (read, write, execute) so that it may implement its various protection mechanisms. Obviously, we don't want to intercept every single operation; instead, the hypervisor masks off entries in the effective page table. Thus, under Bodyguard, the effective page table in the host is a subset of the guest page table defined by the untrusted OS.

Sometimes, an entry cannot be immediately used by the current entity in any form; these entries are masked off entirely. Other times, only certain types of permissions may be masked off, such as masking off the execute permission from certain code pages. The hypervisor will receive page faults when masked-off operations are attempted, or the masked-off pages are accessed; it will perform the proper fixup routines (see the logical page state machines above) to make the access possible.

When the hypervisor is implemented as a component inside a traditional VMM, masking can be performed while the VMM is converting the guest page tables into the host page tables. Such an implementation probably needs to perform eager updating of the masks. The VMM may also choose to cache two copies of certain page tables; one that represents valid accesses by protected code, and one that represents untrusted code.

However, our current implementation (based off the Bochs x86 emulator) uses an alternate approach. Our code lazily generates masks as page table entries are imported into the TLB. When the CPU jumps into or out of a protected process, the hypervisor simply tells Bochs to flush the TLB. This ensures that TLB entries present in the processor are only those that are valid for the current running code.

TLB-fill time is also our opportunity to fixup the frame (if required) before an access is attempted. For instance, if the frame contains a signature but we need to show the private data, we do this during the TLB-fill operation. Or, if the frame contains private data and untrusted code it accessing it, we swap in the signature during the TLB-fill operation.

Finally, TLB-fill time is also our opportunity to unmap pages from a frame when untrusted code is attempting to modify the frame.

4.9 Detecting Memory Corruption

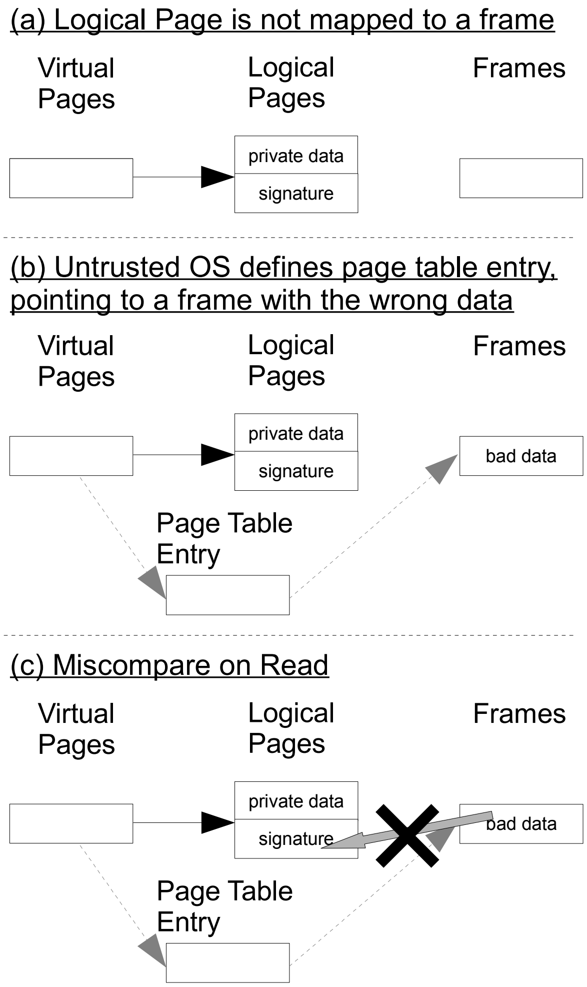
When code attempts to perform any access (execute an instruction, read, or write), this access will happen at a particular virtual address. This access is represented by a page table entry, defined by the untrusted OS, which indicates the frame that stores this page. If the CPU is running a protected process, then this virtual address is also represented by a virtual page data structure in the hypervisor. The virtual page structure points to a logical page, which points to either a frame, or to metadata indicating the stored contents of the logical page.

In the simplest case, the page table and the logical page both indicate the same frame. In this case, the frame is already known to contain the current contents of the page (or, a signature representing said contents). No verification is required.

In another case, the logical page may not be associated with any frame. The contents of the page-table-specified frame are thus compared with the either the saved contents (in the case of a public page) or the signature (in the case of a private page). If the two are exact binary matches, then the hypervisor may associate the logical page with the frame; if not, then corruption has been detected (see Figure 14 below) and the entity will be destroyed.

Finally, it is possible for a logical page to be associated with some **other** frame than the one specified in the page table. (One example is a recent COW-copied page.) In this case, the hypervisor must unmap the logical page from the old frame and then map it into the new frame. Of course, when the hypervisor maps it with the new frame, it must perform the same content verification as we discussed in the previous paragraph.

Thus, we see that corruption is always detected while the hypervisor is attempting to associate a logical page with a frame; once the association exists, the hypervisor trusts the frame to accurately keep track of the contents.



4.10 Register Protections

In order to protect registers, the hypervisor saves their contents and then wipes them each time that the CPU jumps from a protected process to untrusted code. The hypervisor automatically implements this protection so that it will be functional in all types of events, including branches, syscalls, exceptions, interrupts, or breakpoints. Likewise, when the CPU jumps from untrusted code back into the protected code, the hypervisor automatically confirms that the registers match the (wiped) state that is expected, and then restores the old saved values before allowing the operation to proceed.

This protects the application against both corruption and snooping. First, it prevents snooping by wiping the register values, so an attacker cannot interrupt the program as it runs to steal critical data. Second, it prevents corruption by saving the registers as the CPU leaves the protected application and restoring them when it returns. Thus, the operating system is unable to change the state of the process.

Isn't register verification redundant?

If registers are saved and restored, then why does hypervisor check that the registers have the correct values after returning from the operating system? In truth, this is not absolutely necessary. However, with Bodyguard, we chose to implement this for two reasons. First, it constitutes a sanity check that our protected application is coded correctly; if an innocent operating system might modify some registers which the application believes should be saved, then it probably indicates that the programmer does not actually understand the semantics of the operating system completely. Second, it acts as a detector for attackers; as with memory, we judge that if the attacker has attempted to corrupt a protected process once, it is likely to do so again. Thus, we require the hypervisor to verify, when the operating system returns to the protected application, that the registers have the proper (wiped) values before restoring their old values.

The register set

In the paragraphs above, we have described how Bodyguard saves, wipes, confirms, and restores "the registers." The actual set of registers thus protected is, of course, architecture- and operating system-dependent. Stated informally, the registers to be protected, checked, and restored are those registers that the program does not expect will ever change during an interrupt or other non-application-requested event. This includes most or all of the general purpose registers, floating point and other miscellaneous registers, segment registers (if any), and most bits of most machine state registers. Moreover, since different operating systems may have different conventions, we recognize that the list of registers should probably be configurable (controlled, most likely, by the client). Since our current project is limited to Linux on the x86, however, we don't have to worry about this complication.

(Note that the list of registers that should be wiped is likely to be a little shorter than the list to be protected; probably, any application can afford to leak the instruction pointer and stack pointer, at least.)

Additionally, when the application actually performs a syscall, the list of wiped, confirmed, and restored registers will be even shorter. Many syscalls will pass some arguments in registers; most will expect a return code, at least. Before it makes a syscall, a protected application must perform a hypercall, which tells the hypervisor which registers to wipe, and which to confirm and restore on return. Note, however, that this request does **not** automatically apply to the next time that the process leaves protected code. (If it did, an attacker could snoop on variables, in this window of time, using a breakpoint.[20]) Instead, the request applies only to the next time that the process actually performs an explicit syscall.

Signals

Our current design is not able to handle signals, since, with a signal, the kernel alters (at least) the IP register, and probably others. We hope to address this in our Future Work.

[20] The idea here is that the protected process will first make a hypercall informing the hypervisor that a syscall is forthcoming, then initialize the argument registers, and finally make the call. Thus, there is a finite, non-zero amount of code which executes between the hypercall and the syscall.

Imagine that the "allow registers X to pass through next syscall" affected the next transition out of the kernel, and was not specific to syscalls. An attacker could then set up a hardware breakpoint immediately after the return from this hypercall. When the breakpoint fired, the attacker would be able to snoop on registers which were not yet filled with syscall arguments; they would have lingering information from earlier in the program.

By limiting this hypercall to only applying to syscalls, we allow the untrusted OS to interrupt the program arbitrarily many times in the window between the hypercall and the syscall; registers are protected during each such interrupt. Only when the program actually performs the syscall (implying that the registers have been initialized to their syscall-argument values) does the hypervisor allow the registers to leak.

4.11 Syscall Handler

A *syscall handler* is a function inside a protected process that implements wrappers around some syscalls. For instance, it implements bounce buffers for certain syscalls. The syscall handler is registered with the hypervisor during process initialization; thereafter, all syscalls that occur inside the protected process (except those preceded with a "allow next syscall to proceed" hypercall) will get redirected to the syscall handler.

When the hypervisor forces the program into the syscall handler, it pushes an ordinary stack frame onto the stack; the arguments pushed in that frame are the various syscall arguments that were stored in registers. The intent is that it appear to be an ordinary function call into the syscall handler. The syscall handler may thus perform its work, and then deliver a return code simply by returning from the function.

Note that when the hypervisor drives a syscall into the syscall handler, this does is **not** count as jumping out of the protected process. Instead this is a jump **within** the protected process. The syscall handler runs as protected code, and has full access to the private data of the process; also, registers are **not** saved or wiped during this call. Likewise, there is no need to verify the registers when the syscall handler returns, since it is already running protected code.

4.12 Shim

We have implemented a *shim*, which is a shared library that may be linked into a legacy application. While a careful port of a process is preferable and more reliable, this shim is able to implement (almost) complete protection of a legacy application.

The heart of the shim is the syscall handler, which functions as a wrapper around each Linux syscall. Most syscalls are trivial; many can be passed through directly (because they do not include any pointers). Others require only trivial bounce buffers. A handful have difficult semantics or require multiple bounce buffers. However, few stretch more than 20 lines of code.

NOTE: Overshadow (see Chapter 5) also uses a shim, although they implement it as a launcher program rather than a shared library. Additionally, their shim is much more complex, as it has to perform many duties that are handled implicitly in our work.

4.12.1 Limitations

While the shim can implement most of the protection required, there are a few details that it simply is unable to automatically handle. Despite these limitations, the shim is still useful; if the protected process is only slightly modified to handle these situations, then the shim can be used to automatically supply the balance of the code necessary.

Private IPC

We have mentioned private IPC previously in this document. When two protected processes from the same entity need to communicate through an IPC channel, they need to implement some sort of mechanism (such as encryption) to ensure the privacy of the channel. However, in order to use private IPC, the shim must be able to detect that the process at the other end is a protected process. Without this information, the shim will be forced to send all data through normal IPC mechanisms, which, depending on the application, might be a huge hole in the protection.

The shim includes functions that allow the program to designate certain IPC channels as private IPC channels. Once the channels are so designated, the shim will automatically implement protection on this channel.

Shared memory issues

As described above, the protected process must explicitly tell the hypervisor about every virtual page that is a shared page. With some forms of shared memory, it's easy to see what is shared. For instance, the shim can automatically handle `shmget()` and `shmat()` calls, and make sure that the hypervisor is notified about the shared pages. However, other types of shared memory are difficult to detect. For instance, if two processes both map a set of pages from disk, or if the process maps the same disk page into two different virtual addresses, it may be quite difficult to realize that the pages are shared.

The shim includes functions that allow a process to declare that some pages are shared, and how. Once declared, the shim will automatically update the shared page table and notify the hypervisor.

Verifying mmap()s

As discussed above, when new pages are created in the working set, the hypervisor trusts the initial contents of those pages to be correct. The initial contents may be a private page (if the initial frame contains a signature) or a public page. However, the contents of that page may not be what the process expects. For instance, newly allocated pages for the heap (generally allocated with MAP_ANONYMOUS) should be all zeroes, but an attacker could fill them with garbage. Likewise, newly mapped pages from a file should have the correct contents (signature or public data), but an attacker could corrupt this.

(Note that this problem is particularly bad when `mmap()`ing new executable pages. Presumably, the program, when it `mmap()`s an executable page, intends for that page to be part of the protected application. However, if the attacker maps invalid information, the page may be viewed as a public page, and thus not trusted code; if the attacker maps a signature, but the *wrong* signature, then the attacker can actually force the process to execute the wrong code. Thus, `mmap()` verification is particularly critical for executable pages.)

To partially solve this, the shim automatically verifies that MAP_ANONYMOUS pages are filled with zeroes, but otherwise generally does not perform any verification. However, in Future Work, we plan to add a capability wherein the protected application could inform the shim that a file was in a special format, which allows the shim to automatically verify checksums on all mmap()s of that file. Likewise, we hope to add mechanisms that would allow the process to automatically write out such files, so that temporary files can be written and then later verified when they are read.

exec()

Generally, the shim assumes that when `a` protected process creates a child process with `fork()` and then calls `exec()` in the child, it is executing another protected process, such as setting up an ssh tunnel. However, a protected process might occasionally do things in the untrusted OS that involved non-trusted applications. Thus, before a protected process calls `exec()`, it must inform the shim whether the new process should be a protected process or just a normal untrusted one.

The protected process must also specify the initial working set of the newly-`exec()`ed process, as it does with the bootstrap process described below.

4.13 Entity Bootstrap

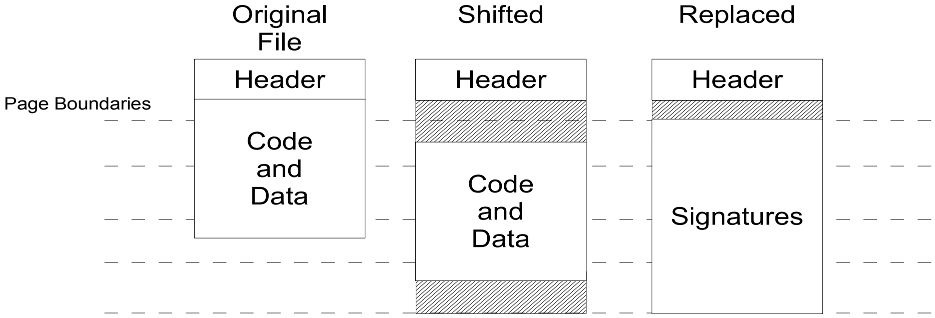
In the above sections, we have detailed how to `fork()` an existing protected process and then handle `exec()` to create a new process. However, we also need a method for verifying the initial state of the first process of an entity.

The basic idea of the bootstrap process is to first modify the executable (and its linker, and also any shared libraries that it links to) such that the code & data use signatures instead of the private values. The client then calculates the starting image it expects for the process, including all of the code and data pages that the OS will automatically map. Next, the client connects to the hypervisor, creates a new entity, uploads all of the signature pages (both the signature values, and the private code and data pages associated with them), and informs the hypervisor about the expected initial page layout. Finally, the client connects to the untrusted OS, uploads the modified files, and runs the program.

The untrusted OS maps the program into a newly-created process, and attempts to run the code. The hypervisor recognizes that the code page is a signature from the newly-created entity; thus, the hypervisor views this as an attempt to jump into a protected process. The hypervisor verifies the register state and the contents of this first code page, and then sets up the working state of the new process object to match the expected layout, which the client communicated previously. From this point on, the process is treated just like any other protected process.

In the sections below, we will detail each step of this process.

4.13.1 Modifying an Executable File



When the client modifies an executable file (a program or shared library), it splits the file into two parts. First, the header[21] is left in cleartext. This is the information that the untrusted OS must read in order to properly map all of the initial code and data pages. Second, all of the code and data is moved out into other pages in the file, and all of those pages are replaced with signatures. The result is a file that looks like a normal executable to the untrusted OS, but when the code and data pages are mapped into memory, each page mapped will actually be a signature, which the hypervisor will be able to recognize as belonging to this entity[22]. (See Figure 15: Modifying an Executable for Bootstrap.)

[21] Depending on the file format, the header might be a single section or might be multiple sections. We generally assume that all of these sections are at the head of the file, but they could be scattered throughout the file. In the latter case, the client has the option of rewriting the layout of the file to make it more convenient, or of moving the code and data sections out past the last such section.

[22] Note that if the client runs multiple copies of this same application, each must have its own modified version of each executable. The reason for this is that the hypervisor detects which entity is running by inspecting the code page that runs; thus, if we have multiple entities, then we must have multiple copies of the same code page. There will be multiple

Finally, the client must make sure that the dynamic linker that is loaded into the program is a dynamic linker that it provides and trusts[23]. (This linker, of course, must also be modified by the process described in this section.) The client can ensure that the correct linker runs in two ways. First, it might run the protected application in a "chroot jail," providing the trusted linker at the appropriate relative path. Second, it might edit the program header to refer to the trusted linker directly, using relative paths[24]. Either way, it ensures that a specific linker, uploaded along with the protected application, is loaded by the untrusted OS as the linker for the new process.

Note that the client expects that the untrusted OS will load all pages into their correct virtual addresses. It enforces this, as described below, by telling the hypervisor the expected starting working set for the process, which the hypervisor enforces as the various pages are eventually accessed. Thus, if the untrusted OS incorrectly loads the process, this will be detected whenever some corrupted page is accessed by the process.

4.13.2 Calculate the Starting Image

To calculate the starting image, the client must analyze the program to be run, and the custom dynamic linker executable. It figures out where the first few pages will get mapped by the untrusted OS (provided, of course, that the untrusted OS behaves properly). It also figures out the initial address where the code should run.

Together, these pieces of data will form the initial working set, and initial register state, of the process to be created.

4.13.3 Create the New Entity

To create the new entity, the client connects to the hypervisor using a trusted channel, such as ssh. The client requests the creation of a new entity. It communicates the initial working set and initial register state (as determined above) to the hypervisor. It communicates to the hypervisor all of the signatures, which were created editing the various executables and shared libraries, along with the private data for each.

The channel then goes idle; there will be no more traffic on it (except perhaps for keepalive messages) until the entity is destroyed.

Note that the network connection between the client and the hypervisor **must** stay open throughout the life of the entity. As mentioned above in Section 4.2.1: Destroying an Entity, the channel is used as an indication between the hypervisor and the client that the client believes that the entity is still doing useful work. If the client believes that the entity has finished its work or crashed, or if something

signatures (one per entity) that happen to represent the same private data.

[23] In some operating systems (Linux being one example), the OS is expected to provide a dynamic linker, which is a small executable which handles the internals of the dynamic linking mechanism. It keeps track of the symbol table, resolves symbols on demand, and may mmap() new pages. For ELF executables on Linux, this is traditionally named `/lib/ld-linux.so.2`.

Obviously, the code in this dynamic linker must be part of the working set of the process, since it will call that code (or read data within it) whenever it calls a function in any shared library. Thus, the dynamic linker must be trusted code, provided by the client, and protected during bootstrap. Otherwise, an attacker could insert a malicious dynamic linker.

[24] ELF executables include an "interpreter" field in their header, which normally points to `/lib/ld-linux.so.2`. However, the header can be edited to specify any file as the interpreter.

interrupts the network connection between the client and the hypervisor, then the connection will close and the hypervisor will destroy the entity. This ensures that the hypervisor is not forced to save old saved private pages indefinitely.

4.13.4 Start the Program in the Untrusted OS

Once the entity is initialized within the hypervisor, the client connects to the untrusted OS and uploads the modified executables and libraries. It then asks the untrusted OS to run the application. The untrusted OS will read the file header of the executable and use that to load the dynamic linker and the required pages of the executable. It then jumps to the start address.

When it jumps to the start address, the hypervisor notices an attempt to execute a code page that contains a known signature. It interprets this as an attempt to execute a private code page. The hypervisor knows which entity is running because the signature is associated with a single entity; it will notice that the entity is waiting for its first process to run.

The hypervisor then verifies the state. First, it verifies that the registers are correct (primarily, that the instruction pointer is at the correct start address). Next, it verifies that the code page being executed is the correct one for that virtual address (as specified by the initial working set uploaded by the client through the secure connection). If both of these are true, then it interprets this as a legal jump into a protected process. It sets up the proper data structures within the hypervisor (particularly, the working set, virtual and logical pages), and allows the process to run.

From then on, the process is treated like any other protected process; once its initial state is correct, corruption and snooping are impossible. Note that while the hypervisor is not able to immediately confirm that the untrusted OS mapped all of the right pages into the right locations, the working set knows exactly what those pages ought to be. Thus, the hypervisor is able to confirm the proper maps as soon as each page is accessed the first time.

4.14 Detailed Examples

4.14.1 Read a Private Page, Inside the Protected Process

Imagine that a protected process attempts to read a certain private virtual page. The page is already `mmap()`ed into the process, but has never been touched. Therefore, the untrusted OS page table entry is not yet populated, and the hypervisor does not yet know about the page. The access requires the following steps:

- The protected process attempts to read the page. Since the page table entry does not exist, the untrusted OS kernel receives a page fault. The hypervisor saves and then wipes the registers as the thread jumps out of protected code.
- The kernel checks and finds that the page has not yet been read from disk. It reads the page from disk. The contents it reads, however, is the signature of the page in question.
- Once the page (its signature, actually) has been loaded into a frame, the kernel defines the page table entry, and schedules the process to run again.
- The hypervisor detects whenever the code jumps back into the protected process, validates the registers, and then restores the values that were saved as the code jumped out.

- The protected process retries the access. This time, although the kernel has defined a page table entry, the hypervisor intercepts the access in order to perform validation.
- The hypervisor checks to see if the virtual page exists in the process' working set. It does not, so the hypervisor defines a new virtual page, links it to a new logical page, and places it in the working set.
- The hypervisor inspects the contents of the frame pointed to by the page table entry. In this case, it contains a signature, so the logical page is initialized such that its saved contents are the old private data.
- Since the protected process is attempting to read the page, the hypervisor replaces the signature with the private data.
- The hypervisor now allows the access to proceed.

4.14.2 Write a COW Private Page, Inside the Protected Process

At some time after a `fork()`, a protected process tries to write a certain page. This page has not yet been modified by either process, and as such they still share the same COW page. The current process has a readonly page table entry for this page, and the frame contains the private data, which is currently being shared by two or more processes.

- The protected process attempts to write the page. Since the page table entry is readonly, this results in a page fault to the untrusted OS kernel. The hypervisor saves and wipes the registers as the thread jumps out of protected code.
- The guest OS kernel determines that this is a COW page. It attempts to read the page so as to copy it to another guest physical page. The hypervisor intercepts the read, and swaps in a signature (generating one, if needed).
- The kernel access may now proceed; it copies the signature to another frame. It then updates the page table to point this virtual address to the new frame, and turns on write access in that entry only.
- The kernel returns to the protected process. The hypervisor detects the transition back to protected code, validates the registers, and then restored the saved values. The protected process now retries the access. Again, the hypervisor intercepts the access.
- In this case, the hypervisor notices that the frame being pointed to by the page table is different than the current location of this logical page. (The hypervisor's logical page struct still points to the old frame.) Since the old frame already contains a signature, the hypervisor simply unmaps this logical page from that frame, and remaps it in the new location. Of course, the hypervisor confirms that the new frame has the expected contents (the signature) before mapping the logical page to the new frame.
- The hypervisor next replaces the signature in the new page with the private data, and finally allows the access to proceed. Note that the old frame will continue to contain the signature until such a time as some protected process actually accesses that frame again.

4.14.3 Syscall

This scenario details how a syscall coming from the protected process is handled.

- The protected process is running. It performs a syscall.
- The hypervisor intercepts the syscall. It pushes a new frame onto the stack, as if the calling code had performed a function call instead of a syscall. The instruction pointer is changed to point to the syscall handler.
- The syscall handler runs, and performs whatever emulation it must do.
- The syscall handler must now call the actual untrusted OS syscall. To do this, it first uses a hypercall to tell the hypervisor that the next syscall should proceed to the untrusted OS, and then performs an ordinary syscall.
- As the code leaves the protected process, the hypervisor stores the register state and wipes any registers that are not syscall parameters.
- The syscall code in the kernel runs. If appropriate, it may read or write pages in the protected process (presumably, bounce buffers were used). It eventually returns; the hypervisor detects the transition back into protected code, and validates and restores the registers as appropriate. (Note that some of the registers will not be validated and will not be restored, because they hold the return value(s).)
- The syscall handler now does any post-processing for the emulation. It finally returns to the caller using a normal function call return. Note that the hypervisor is not involved in this return to the original code[25].
- The code that originally performed the syscall runs.

4.14.4 Swap Out/Swap In

This section details how a page can be swapped out by the guest OS and then later swapped back in.

- While running kernel code, the guest OS may decide that it needs to swap out some pages. The pages that it chooses may include a private page from some application.
- The guest OS deletes page table entries from the appropriate page table(s) and then flushes the TLB on all processor(s).
- The guest OS attempts to write the page to disk. This, presumably, involves a DMA operation. DMA operations are treated as accesses by untrusted code, and so the private data is replaced by a signature before the DMA is allowed to proceed. The signature is then written to disk.
- The guest OS allocates the frame for other purposes. Once untrusted code (or code from other entities) starts writing to the frame, the hypervisor breaks any association between the logical page and the frame, and forgets that the frame used to contain a signature; it is now just a generic frame, containing raw data.

[25] This is the design intent. However, if you read the Implementation chapter, you will find that one of the hacks is that we currently use a hypercall to unwrap the stack. This is simply a temporary measure, because the author didn't have time to investigate the exact format of a correct stack frame.

- Later, the protected process page faults on the virtual address. At this point, this becomes a normal read operation (see the example above); the untrusted OS reads the page from disk and sets up the page table. The access is retried, intercepted by the hypervisor, and the private data restored.

4.14.5 Simple Corruption

This section gives an example of a straightforward corruption attempt; the attacker tries to write to a virtual page.

- A protected process is running. The hypervisor knows the working set of the process. There exists some particular virtual page, which the attacker will soon overwrite; currently, this page is loaded into some frame, and the private data is showing.
- An interrupt or syscall occurs. The attacker, running inside the kernel, decides to write to the victim virtual page. It uses the same virtual address as was used by the protected process. However, this access is intercepted because the current code is untrusted code.
- The hypervisor saves the contents of the frame, and then drops the signature into the frame. Then, because the current access is an attempt to write, all logical pages are unmapped from this frame.
- The attacker is allowed to write to the frame. It then returns to the protected process.
- The protected process runs as normal, for some arbitrary amount of time. Eventually, it attempts to access the virtual page again. The hypervisor attempts to map the logical page back into the frame (since the page table still says that the virtual page resides in that frame), but the frame's contents to **not** match the signature of the logical page.
- The hypervisor destroys the entity, and forces the process to crash.

4.15 Summary

This chapter detailed our protection scheme. It first described the key elements of Bodyguard's design: the hypervisor, entities, protected processes, the working set, virtual and logical pages, and physical frames. It next detailed signatures, mapping guest page tables to host page tables, and register protection. It detailed the shim, including the syscall handler. It detailed how the client and hypervisor initialize a new entity securely. It closed with a set of step-by-step examples of Bodyguard in action.

Chapter 5: Overshadow

This chapter details the Overshadow design, described in their ASPLOS paper [25]. Overshadow is an existing design which has many similarities with Bodyguard. Most elements of Overshadow map relatively easily to concepts from our design[26], but there exist a number of interesting tradeoffs that we will explore.

The chapter begins by detailing the Overshadow design. It discusses the concept of "multi-shadowing" (their abstraction for presenting various versions of a page to various blocks of code), "applications" (entities), "address spaces" (processes), and their thread design.

Next, the chapter describes the concept of a "protected object," which is central to how they track and enforce the working set of their address spaces. It details their shim, which does all of the same work as our shim, but also is responsible for a number other things, including interacting with the VMM to protect the working set of the address space, and implementing a "trampoline" function for handling jumps from untrusted code back into a protected process.

Next, the chapter presents detailed examples of how all of this works in practice.

Finally, the chapter discusses the similarities, differences, and tradeoffs between Bodyguard and Overshadow.

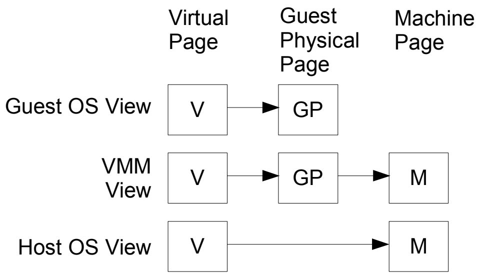
5.1 Shadows

Overshadow is a VMM (based on VMware [26]) that uses multi-shadowing [27], a virtualization strategy where the same guest physical[27] page may map to different contents at different times. Overshadow uses this to protect private data. Each private page is "multi-shadowed:" in one shadow, the page contains the private data; in all other shadow(s) the page contains an encrypted version of that same data.

Modern VMMs such as VMware run applications natively whenever possible. Thus, the host must be configured with a page table that maps virtual addresses (as perceived by guest code) to machine pages. To accomplish this, the VMM watches the guest page table (which is written by the guest OS). Every time that it changes, the VMM updates the host page table to match. Typically, each valid entry in the guest page table should map to a similar one in the host page table. (See Figure 16.)

[26] Note that in the descriptions below, Overshadow terminology is used in most cases. In a few cases, where concepts are identical (such as the concept of a "protected process"), terminology from this thesis will be used. However, the chapter is consistent throughout; a given concept will use either the Overshadow term, or the one from this thesis.

[27] A guest physical page is a "physical" page from the perspective of the guest OS. This is in contrast to a "machine" page, which is a page in the host machine. In a traditional VMM, the mapping of guest physical pages to machine pages is typically one-to-one.



Shadowing adds a wrinkle to this process. Now, if a given guest physical page contains private data, then it has different contents in different shadows. (See Figure 17.) Thus, the VMM maintains multiple copies of the page table, one for each shadow.

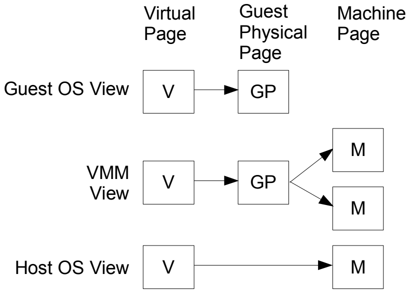


Figure 17: Various Views of the Virtual Space (private page)

In theory, all shadows could have mappings for all guest physical pages at all times. However, this would require that the VMM have two machine pages for each guest physical page (when the guest physical has private data). More importantly, it requires that Overshadow continually re-encrypt the private data, so that the encrypted page stays in sync with the private contents.

Instead, Overshadow implements a single machine page for each guest physical page, and encrypts/decrypts it as access patterns require. This requires that Overshadow mask out certain guest physical pages from certain shadows at certain times. We say that a guest physical page is “mapped into” a shadow if the guest physical page is in a state where that shadow is entitled to read its contents. That is, when a guest physical page contains private data, it is mapped into only the application shadow that owns it; when a guest physical page contains encrypted data (or anything else that is not private data), then it is mapped into the “system shadow,” which represents external access[28]. The page is never mapped into both shadows at the same time.[29]

When code attempts to access a page that is not mapped into its shadow, Overshadow automatically takes the actions necessary to change the page state, unmap it from its current shadow, and map it into this one. That is, if a protected application attempts to touch one of its pages, but that page is currently

[28] The Overshadow paper doesn't directly discuss the possibility that a protected process might try to touch a private page owned by some **different** application. This, of course, would not be a normal access pattern, but might occur as part of an attack. This author presumes that Overshadow would handle this the same way that we do; it would allow the accessor to see the encrypted page, just like any other untrusted code.

[29] The author believes that this is true only for pages which contain private data, but could not find an explicit statement of this in the Overshadow paper.

encrypted, the VMM will automatically decrypt it, unmapping that page from the system shadow and mapping it into the application shadow. Similarly, if untrusted code attempts to touch the same page, the page will be re-encrypted and the mapping returned to the system shadow.

This shadowing system parallels our signature mechanism fairly closely; each system keeps track of whether a physical page has private data or its encrypted version; external code (primarily the kernel) can access each private page, but will see the encrypted version instead of the private version. Pages automatically and invisibly move between the two states, based on accesses.

5.2 Shadowed vs. Unshadowed Pages

Overshadow handles both shadowed and unshadowed pages. Unshadowed pages are the default; these are normal pages, handled with ordinary VMM mechanisms. Shadowed pages are protected pages, containing private data; Overshadow ensures that both corruption and snooping are prevented.

One key difference between Overshadow and our work is that, in our design, all pages in a protected process are private pages by default. Pages may be declared to be public, but they retain that designation only until they are modified; once they are modified, they immediately become private again. However, in Overshadow, each protected process contains a mix of shadowed and unshadowed (that is, private and public) pages. These designations are persistent, meaning that if you write to a public page, it stays public (this is how Overshadow implements bounce buffers). For the same reason, Overshadow does not need a mechanism to declare a private page to be public; all private pages are private, permanently.

5.3 Protected Objects

Overshadow manages protected memory through *cloaked resources*. A cloaked resource is a linear (but perhaps sparse) object. (Examples of resources include files and anonymous memory.) The various pages in the resource are assigned RPNs (Resource Page Numbers). Each cloaked resource is assigned an RID, which is a 64-bit unique Resource Identifier.

The VMM is responsible for keeping track of the state of all cloaked resources. In particular, for each valid RPN in the resource, it stores the current guest physical page that stores this page (if any), and the encryption metadata (if any).

The VMM also has a table, indexed by guest physical page number, which gives the (RID,RPN), if any, for each guest physical page. This makes it possible, when the untrusted code accesses a guest physical page, to determine whether or not a guest physical page contains any private data, and if so, where to store the encryption metadata once the page is encrypted.

5.3.1 Encryption

When the system shadow attempts to access a guest physical page but this page is not mapped into that shadow, the VMM will encrypt the guest physical page[30].

All pages are encrypted with a single private key, known only to the VMM. However, every encryption operation uses a randomly-generated initialization vector (IV); this ensures that dictionary attacks are impossible, since encrypting two identical pages will generate different ciphertext. After

[30] Since we assume that guest physical pages containing public pages (and encrypted private pages) are always mapped into the system shadow, we the VMM knows that any missing page must contain private data.

encryption, the page is hashed. The metadata pair (IV,H) is stored in the VMM tables for the (RID,RPN) pair that was previously associated with this guest physical page.

Once the page is encrypted and the (IV,H) is saved, the page is “zapped” (unmapped from all shadows) and then mapped into the system shadow. The access may now proceed.

5.3.2 Validation & Decryption

(Note that this section does not detail how the VMM looks up a virtual page to figure out what (RID,RPN) should reside there, which will be discussed below. This only details the decryption protocol.)

When a protected application needs to decrypt a page, the VMM looks up the (IV,H) value for this (RID,RPN). The VMM then first confirms that the hash of the physical page matches the saved H value. (If it does not, then this constitutes corruption, and the protected process can be killed.) However, if the hash is valid, then the VMM will decrypt the page, using the VMM private key and the initialization vector IV.

Once the page is decrypted, the page is “zapped” and then mapped into the application shadow.

The (IV,H) pair are retained in the VMM's metadata so long as the page is readonly. If it is later necessary to encrypt the page again, it will be encrypted with the same IV, thus ensuring that the re-encrypted version is exactly the same as before. This is necessary for correctness. (See Section 4.7.3: Signatures and Readonly Pages.)

The (IV,H) value for the (RID,RPN) is discarded when the page is modified; in that case, a new IV will be generated (and a new H calculated) the next time that it becomes necessary to encrypt the page.

5.3.3 Discarding Objects

Some objects may be automatically discarded when not in use. For instance, objects that represent private non-serialized data of a process (stack, heap, anonymous memory, etc.) can be deleted when the process dies.

However, some other objects must persist. For instance, files that are written to disk and that contain private data should be readable by other processes in the same application.

5.3.4 Serialization of Metadata

Resources that are actively in use have their metadata stored in VMM memory. This metadata includes the list of valid pages within the object, and the metadata for each such page. (The metadata states that the page currently resides, cleartext, in some guest physical page and/or indicates the saved (IV,H) pair for this private page.)

However, some resources may exist that the VMM knows about but are not actively in use. The VMM flushes the metadata for these objects out to the guest OS disk. Each resource is represented by a single file on the guest OS disk; this file contains all of the metadata for the protected object. The contents of these files are encrypted, and include a SHA-256 hash, which ensures that the contents are correct when they are later read from disk. (The details of exactly how this is handled are not entirely clear.)

The VMM writes out files to guest OS disk using a daemon process (osfd) running inside the guest OS;

this daemon is not trusted, and so only handles encrypted pages coming from the VMM. The VMM reads files back from disk (when a resource needs to be restored) via the shim. After reading the file up from disk, the VMM validates the file using the hash and decrypts its contents[31]; it may then use the metadata to determine the valid RPNs and the stored (IV,H) values for each.

5.3.5 Serialization of Contents

The section above describes how the VMM may serialize the metadata of a protected resource to disk. However, this does not serialize the contents; it simply serializes the list of valid pages and the table of (IV,H) pairs. This makes it possible for the VMM to validate the contents of the pages (preventing corruption), but it does not inform the guest OS that it is possible to write these pages to disk.

For pages that the guest OS can write to disk (that is, the actual contents of certain resources), the shim uses `mmap()`ed pages (always `mmap()`ed with MAP_SHARED so that different processes always see the same contents). These pages map to actual files on the guest OS disk, and the guest OS may flush them to disk if desired. The file contents include a header, the raw data, and some padding. (See Section 5.4.3 below.) Since these pages are private pages, with all of the normal protections, these pages will be encrypted before they are written to disk, and the VMM will be able to confirm their contents after they are read back.

Thus, opening a new file from the guest OS disk can involves as many as 3 basic operations. First, it opens the file inside the guest OS and maps the contents of that file to the virtual memory of the process. Then, it tells the VMM which protected resource represents this file[32], and maps the RPN numbers to the virtual addresses. Finally, (if necessary), the shim may need to load the metadata file from disk into VMM memory. Once the shim has done both of these, the pages may be safely used; the VMM knows what contents to expect for these pages, and can validate their contents.

5.4 Shim

Overshadow uses a shim that, at a very high level, is similar to the shim in our design. The purpose of the shim is to provide a wrapper that automatically wraps a legacy process and automatically implements the protections necessary.

Their implementation, though, is very different from our shim. First, their shim is split into shadowed and unshadowed parts; the unshadowed part does not contain any private data, but it implements a trampoline mechanism, which allows Overshadow to detect when the guest OS is attempting to return us to a protected process. (In our design, no trampoline is required, because the hypervisor can detect a return to the protected process automatically.) The unshadowed part of the shim also contains the bounce buffers, which are used when the process needs to give a buffer to the kernel for writing or reading.

Second, their shim explicitly tracks the virtual space, which is not required in our design, because the VMM can implement this implicitly. The shim must eagerly keep track of every `mmap()`, `munmap()`, `brk()`, and also all of the implicit map actions (such as stack growth). It communicates the expected state of the virtual space to the VMM (as detailed later), which gives the VMM the information necessary to prevent page corruption.

[31] Or does it decrypt it, and then verify the hash? This author is not certain.

[32] This author is not clear how the shim determines this. Perhaps the file header, which is tacked onto the front of the file, includes the RID???

Third, their shim implements bootstrap within the guest OS, while running unshadowed. That is, their shim (running unprotected) runs as a normal process within the guest OS. This reads the executable and loads it manually into the protected space. This mechanism (as acknowledged in their paper) is obviously insecure if the attacker has compromised the machine before the shim runs. (Our bootstrap process is secure even if the guest OS has been compromised long before the client starts our new entity.)

5.4.1 The Trampoline

The shim implements a "trampoline" mechanism to handle transitions from untrusted code to a protected process. The trampoline is a snippet of code in the unshadowed portion of the shim. Since it is unshadowed, it may only contain public data, (no private secrets), and nothing it does can be trusted. However, it may perform hypercalls into the VMM and make requests.

When the trampoline runs, it hypercalls into the VMM, giving an ASID (address space ID) and a CTC address. The ASID is a public value that uniquely identifies the address space; the CTC address is the virtual address of a private page that contains the CTC for the thread that the kernel has selected to run.

When the VMM receives the hypercall, it cannot trust that the call is valid; however, it will attempt to execute the thread specified. To do this, it first looks up the ASID and validates it; it then attempts to read the virtual page. Like any access to a private virtual page (which will be detailed later), this means that the VMM looks up the page table entry for this virtual address, compares it with the expected contents that the VMM knows for this virtual address, validates the page, and decrypts it if necessary. If this process fails (for instance, the page table entry doesn't exist, or is readonly), then the VMM will return an error to the trampoline, which will touch the page (causing a page fault) and then retry. Once the page is writable, the VMM can read the contents of the CTC, stored in the virtual page.

The CTC first includes the ASID and the private address space identifier. This validates that the page is actually a member of the desired application, and not something created by an attacker. The rest of the CTC contains the saved thread state; the VMM reads this thread state into the registers, and then the thread may run.

The VMM keeps track of the current address space, and the CTC of this thread; when the CPU finally jumps out of the protected process for any reason (interrupt, exception, breakpoint, syscall, etc.), the VMM will write out the new thread state to the CTC. Before actually allowing untrusted code to run, the VMM also wipes most registers, in order to prevent information leaks. Finally, the VMM changes the IP and SP to new values, which point to the trampoline code. Thus, when the code jumps into the kernel, the kernel believes that the process was still executing the trampoline code; when it finally schedules the thread to run again, it will return the process to that code. The trampoline code will then run once more, and go through the work of activating the thread again.

Note that the trampoline mechanism does not require the VMM to know how many threads exist. Instead, it provides a mechanism for validating attempts to restore thread state, where a trampoline action only works if it points to a valid CTC. By creating new CTCs (or deleting old ones), the protected part of the shim makes it possible for new threads to be created, or to delete old ones.

5.4.2 Syscall Emulation

As stated above, in most transitions to kernel space, the thread state is automatically stored to the CTC, the registers wiped, and the IP/SP altered before the code jumps into untrusted code. However, syscalls are handled specially. As with our design, the Overshadow shim includes a syscall handler, which intercepts syscall attempts, redirecting them to the protected shim. While the registers are all saved, they are not wiped in this case, and code does not jump to untrusted code. Instead, the VMM forces the code into a specially designated function in the shim.

As with our code, the syscall handler generally will call the guest OS after performing some amount of emulation (sometimes a lot, sometimes little); this second syscall will actually be passed on to the kernel (with some amount of the registers **not** wiped, since they carry arguments). For this reason, the CTC actually contains enough buffer space for two copies of the registers; the first is used for the first syscall, which is redirected to the syscall handler; the second is used for the syscall handler's call into the kernel. These are, of course, used in reverse order when the trampoline code returns to the protected process.

The syscall handler in the Overshadow shim performs syscall emulation, similarly to how our shim does. However, there are several noteworthy differences.

First, the Overshadow shim uses bounce buffers in the "uncloaked" portion of the shim. These are permanently public pages. (By contrast, our design temporarily designates pages as public in order to send data to the kernel, and removes them from the working set when receiving data from the kernel.)

Second, Overshadow's shim performs complex emulation of some operations. In particular, all file operations, such as `read()` and `write()`, are converted into loads and stores on `mmap()`ed buffers, as will be detailed in the next section.

Third, the Overshadow paper claims that their shim provides complete emulation of all syscalls. This author believes that this is impossible, at least for some corner cases. For instance, if a process attempts to open a file for reading, does it expect the file to be a private file, with protected contents? Or is it a public file, to be read raw? Likewise, if it opens a file for writing, should the contents be written cleartext or encrypted? Similar problems exist with pipes to other processes (is the other process in this application or not?), shared memory, etc.

This author believes that "almost perfect" syscall emulation can be achieved, but that truly perfect emulation, correct in all use cases, is impossible.

5.4.3 File Emulation

All file operations on all files[33] are emulated by the shim. When a process holds a file open, the shim `mmap()`s certain pages into private memory. The most important of these is the first page, which holds the file header. This header gives the true file length[34] and other necessary information. The shim implements all operations, notably `read()` and `write()`, as loads and stores on `mmap()`ed pages.

Since it is possible for multiple processes within the application to have the same file open at once, all pages are `mmap()`ed as MAP_SHARED, so that each process will see the same contents. The processes must, of course, then work together to keep the file state consistent. (The exact details of this are not given in the Overshadow paper.)

[33] Since the Overshadow paper doesn't mention any ability to handle raw files, this author will assume that all files are handled as private files.

[34] The file length, as seen by the guest OS, is always rounded up to a multiple of 1MB. Overshadow does this so that they can always mmap() file contents in 1MB blocks, amortizing the cost of these operations.

Note that all such files are protected resources, as discussed above. Thus, when the shim opens a file from the guest OS, it tells the VMM to load the resource metadata for that protected resource. When it maps any page (including the header), records the (RID,RPN) for the virtual page. Once this association is defined, the VMM can verify the contents of the page. Thus, as the shim `mmap()`s and `munmap()`s various pages from the file, the VMM can always ensure that the contents of what gets `mmap()`ed match the expected contents of that protected object. This also allows it to make sure that the various processes all see consistent state; if two processes map the same (RID,RPN) for certain pages, the VMM can easily confirm that both see the same private contents.

5.5 Virtual Memory Protections

The previous sections have discussed, in part, how the VMM is able to ensure that each virtual page has the correct contents. This section will discuss it in detail.

The shim is responsible for keeping track of what virtual pages exist in the process's working set, and the proper (RID,RPN) for each. It tracks all `mmap()`, `munmap()`, `brk()`, and similar syscalls; it also keeps track of stack pages and such.

The VMM stores a cache of the list of virtual pages known to the shim. When there is a cache miss, the VMM performs an upcall into the shim, asking it to deliver the correct mapping. The shim responds with a hypercall that gives the correct (RID,RPN) mapping (if any) for the virtual address.

The guest OS maps virtual addresses to guest physical pages. If the guest OS is working correctly, the pages that it maps to will always contain either the correct private data for the virtual address, or else the encrypted version of the same. Of course, the guest OS might have been compromised by an attacker, and thus might be giving invalid contents.

Mapping & Confirmation Process

In order to map a virtual address to a guest physical page, and to confirm the correct contents of that page, the VMM first looks up the page table entry. If the access is not allowed, then the VMM immediately signals a page fault to the guest OS, and no further validation is required. However, assuming that the page table entry allows the access, the VMM must map the virtual address to an (RID,RPN), map that to expected contents and then compare that to the actual contents. (See Figure 18.)

The first step is to see if the VMM knows about the virtual address. If not, it upcalls into the shim in order to populate the tables (as discussed above). Once it knows about the virtual page, then it knows the (RID,RPN) for this virtual page.

The VMM next indexes into the metadata for the protected object, looking for the metadata for this particular page. It will find either that the page is associated with a current guest physical page or that the (IV,H) value is known. (In the case of decrypted but readonly pages, the metadata will know both.)

Next, the VMM looks up the guest physical page defined in the page table entry. There are several possibilities that can happen now:

- The private page is known to be associated with this guest physical page already. In this case, the guest physical contains the private data, and the access can proceed without any trouble.
- The private page has only the (IV,H) value. This means that the guest physical page must contain the encrypted version. The page is validated (using the hash value H) and then

decrypted (using the initialization vector IV).

- The private page is associated with a **different** guest physical page. (This is rare, but can happen, for instance, with a recently copied COW page.) In this case, the association with the old guest physical page is broken (encrypting first if required[35]), and then the value in the new page is confirmed and decrypted, exactly as in the previous case.

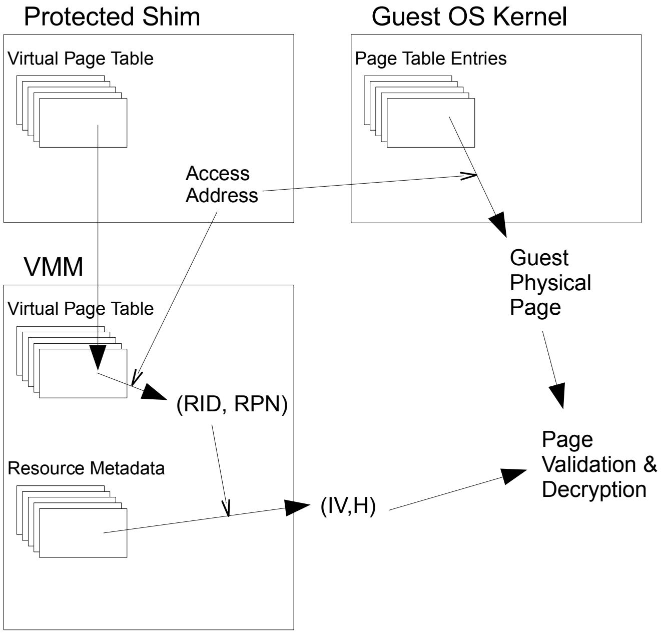


Figure 18: From Virtual Address to Page Validation

5.6 Applications

The Overshadow VMM keeps track of the various "applications" that exist. An application represents a single protection domain, containing one or more protected processes. The various processes may share their private pages with each other, but these pages will be protected from outside access. Each application has a dedicated shadow, which represents the guest physical memory as viewed by these processes.

An Overshadow application is thus an almost perfect match for an "entity" from this thesis.

5.7 Address Spaces

Each application has one or more address spaces associated with it. Each address space represents the virtual memory space of one process.

A shim inside a protected process maintains a table of all known virtual page mappings in that process. It is responsible for keeping the VMM aware of each of these mappings, updating the VMM as pages are `mmap()`ed, `munmap()`ed, or remapped. The VMM uses these mappings to validate pages that the guest OS attempts to map into the address space of the protected process.

In addition, each address space has two assigned identifiers. First, the ASID (Address Space ID) is a public value (shared with untrusted code) that uniquely identifies this address space. Second is a randomly-generated private number that is stored in the protected process' private memory (in the CTCs, see below) but that never should be shared with untrusted code. The combination of these two elements are key to the "trampoline" code, which runs in untrusted code and informs the VMM when the kernel wants to run the protected process again.

5.7.1 fork()

When a protected process is `fork()`ed, the VMM must figure out the correct (RID,RPN) values for

[35] The VMM encrypts the page as it breaks the association in order to prevent lingering private data which might leak to other processes.

the various virtual pages in both processes. Some pages represent shared objects, and thus both processes simply need to keep the same (RID,RPN) value as before. However, anonymous pages (or other MAP_PRIVATE pages, such as MAP_PRIVATE `mmap()`s of files) must be split.

Unlike our design, Overshadow does not immediately split these pages. Instead, it keeps trees that represent the parentage of these pages. That is, so long as two pages are still COW copies of each other, they will share a single (RID,RPN) value. However, upon the first write to either page, Overshadow splits them into two pages with different RPN values.

While the concept of late copying of logical pages is fairly simple, the implementation described in the Overshadow is quite complex. One of the key issues that they must face is that the protected shims (in several processes) must interact to properly implement this COW functionality.

5.8 Threads

Threads are not tracked by the VMM. Instead, the shim maintains a CTC (Cloaked Thread Context) structure in process private memory for each thread. Each CTC contains a secret to validate that this is a CTC (the ASID of the address space, plus the private random number of this ASID). More importantly, it also contains space to save the contents of all of the registers.

When a process jumps out of protected code for any reason, the thread state is stored in the CTC, in *virtual memory*! The VMM can trust that the page exists because the CTC is also used for the "trampoline" code (see below); the VMM fails the trampoline if the virtual page containing the CTC is not present and writable. Since the OS cannot flush any page entries until after the code leaves the protected process[36], the VMM can trust that a page that is writable when code trampolines into protected code will still be writable when it leaves.

The very first CTC in a process is initialized by the VMM during process initialization. The VMM fills in the ASID and the address space's private number. However, later threads (if any) will be initialized by the process itself. It will create the new CTCs by copying the parent CTC and then editing a few fields (in particular, the instruction pointer and stack pointer). Once the new CTC is initialized, the process may ask the untrusted OS to create the new thread. Whenever the new thread runs, it will trampoline into the new thread; the VMM knows that the trampoline is valid because the virtual address pointed to has the right ASID and address space private number. With those verified, the VMM will restore the thread state stored in that particular CTC.

Thus, the VMM has no need to know how many threads exist in each process; nor does it keep track of their state when they jump to the kernel. Instead, the VMM has a mechanism to validate the CTC structure when a trampoline asks to validate a thread stored there. While running, the VMM knows the active address space, and the virtual address of the CTC where thread state should be stored back to; once the code jumps back into the kernel and state has been saved to the CTC, the VMM can forget that

[36] This assertion is based on two assumptions. First, that the architecture has a TLB; second, that the guest OS has no way to flush the thread's TLB without first jumping into kernel code. If both of these are true, then the kernel must know that a page table entry, once defined, may sit in the TLB and could still be in use until we jump back to kernel code and flush the TLB. If the kernel knows this, then it cannot expect any alteration or invalidation of the page table entry to take place until after the TLB flush from kernel code. Thus, the hypervisor (acting on behalf of the protected process) may legitimately write through the virtual page to the guest physical page at the moment when we jump out of the protected process. This write is no more or less valid than any other write which might happen through a cached TLB entry.

If an architecture violates either of these assumptions, then Overshadow does not mention a solution. However, since they only run on VMware (which x86 only), it is not a major issue for them.

this thread ever existed.

Signals

Overshadow uses a variant of the trampoline mechanism to handle signals. Each signal registered by the protected process is intercepted by the shim and stored in a private table. The shim then registers a signal handler with the guest OS, which points to code in the uncloaked portion of the shim. (This signal handler functions as an alternative trampoline.)

When a signal occurs, the signal-handler-trampoline hypercalls into the VMM, passing it the address space ID and CTC address (just like the normal trampoline), but also passing the signal metadata, such as signal number, IP when interrupted, etc.

The VMM uses a combination of CTC information and the signal metadata to determine what was running when the signal occurred. If the protected application was running, then the VMM forces the thread into a signal handler in the protected shim, which forwards the signal to the proper handler in the application. If the protected shim was running, it chooses (based on a flag in the CTC, presumably set by the shim) to either roll the state back to the most recent syscall entry point, or defers signal handling until the shim returns to the protected application.[37]

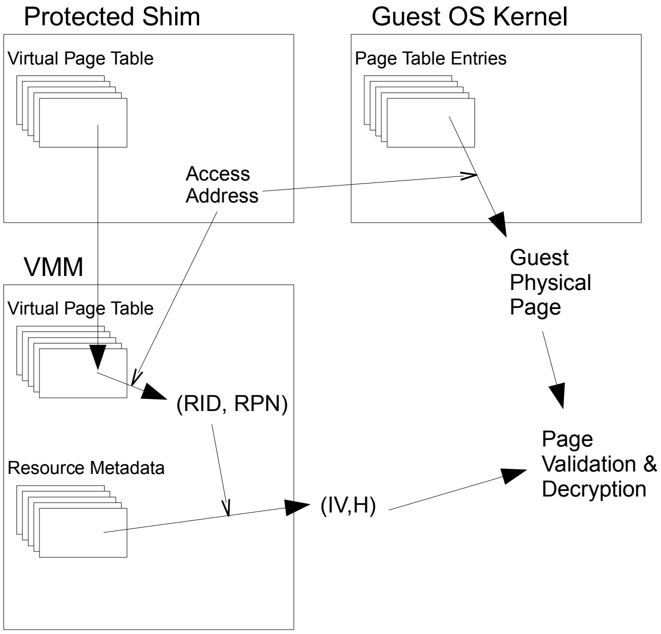


Figure 19: From Virtual Address to Page Validation

5.9 Detailed Examples

5.9.1 Read a Private Page, Inside the Protected Process

Imagine that a protected process attempts to read a certain private virtual page. The page is already `mmap()` ed into the process, and the shim's tables know about it, but this is the first use, so the page table entry is not yet populated, and the VMM does not yet know about the page. The access would requires the following steps:

- The protected process attempts to read the page. Since the page table entry does not exist, the VMM sends a page fault to the guest OS kernel. The thread state is saved in the CTC.
- The guest OS kernel checks and finds that the page has not yet been read from disk. It reads the page from disk. The contents it reads, however, are the encrypted version of the private page.
- Once the page (its encrypted version) has been loaded into a guest physical page, the guest OS defines the page table entry, and schedules the process to run again. Note that while the guest OS has defined a page table entry, the host page table does not yet have a mapping for this virtual address.

[37] Note that the Overshadow paper does not discuss how to handle signals that hit when running uncloaked code.

- The trampoline code in the thread runs, and calls the hypervisor to re-activate the thread. The hypervisor reads the CTC, verifies it, and then restores the thread state.
- The protected process retries the access. Since the host page table doesn't know about this virtual address, a page fault to the VMM occurs. However, it sees that the guest OS page table has a mapping, so it decides to handle this page fault inside the VMM.
- The VMM looks up the virtual address in its table for this address space, and finds that it does not have an entry for this virtual page. It upcalls into the shim, asking for information.
- The protected shim runs, and looks up the (RID,RPN) value for this virtual address. It hypercalls back to the VMM to provide this information, then returns from the upcall.
- The VMM, now having the (RID,RPN) value, looks up the metadata for this page. It finds that the page is not presently in any guest physical page, but that it has a stored (IV,H) pair for the page.
- The VMM looks up the to see which guest physical page is indicated in the guest OS page table.
- The VMM hashes the guest physical page, and compares it to the hash value H. The hash matches.
- The VMM decrypts the guest physical page (using the initialization vector IV) and associates it with this (RID,RPN) value. It zaps the guest physical page and maps it to the application shadow of the current process.
- The VMM adds a host page table entry for this virtual address, pointing to the machine page that stores this guest physical page.
- The VMM returns to the protected process. It retries the access, which now works.

5.9.2 Write a COW Private Page, Inside the Protected Process

Imagine that a protected process attempts to write a certain private virtual page. We will assume that the page is a private COW page. The guest OS knows that this, and other virtual pages, are COW copies of each other. When this sequence begins, we assume that the guest OS page table and the host OS page table both have mappings for the virtual address, in readonly mode. The guest page is mapped into the application shadow.

- The protected process attempts to write the page. Since the page table entry is readonly, the VMM sends a page fault to the guest OS kernel. The thread state is saved in the CTC.
- The guest OS kernel determines that this is a COW page. It attempts to read the page so as to copy it to another guest physical page. The read, however, is not allowed, because the guest physical page is currently mapped into the application shadow. Thus, a page fault to the VMM occurs.
- The VMM detects that code outside the application shadow is attempting to access the page. It encrypts the page, saving the (IV,H) value for this (RID,RPN). It then zaps the guest physical page from the application shadow and maps it into the system shadow.
- The VMM returns to the kernel code, which retries the access. Since the page is now mapped

into the system shadow, this access succeeds. The kernel copies the encrypted data to another guest physical page.

- The kernel updates the guest OS page tables such that the virtual address points to the new guest physical page; this new entry has write permissions. The kernel then schedules the thread to run again.
- The trampoline code runs, and calls the hypervisor to return to the protected process. The VMM validates the CTC and then restores the thread state.
- The protected process retries its access. While the guest OS page table has an entry for this virtual address, the host page table does not, so another page fault to the VMM occurs.
- The VMM goes through the same validation process as we discussed above, in the read example. It ends up decrypting the (new) guest physical page and mapping it into the system shadow. Note that the old guest physical page is still encrypted.
- Before the VMM returns to the protected process, it notices that the access being attempted is a write. It also notices that the (RID,RPN) in question is a COW copy (probably arising from a `fork()` that happened a while ago). It realizes that it must now perform page duplication (see the section on `fork()` above), so it allocates a new (RID,RPN) for this virtual address.[38]
- The VMM returns to the protected process, which retries the write. This write now succeeds, because the guest OS page table has a writable entry for this virtual address, and this entry has been validated and propagated into the host page table.

5.9.3 Syscall

This scenario details how a syscall coming from the protected process is handled.

- The protected process is running. It performs a syscall.
- The VMM saves the current thread state to the CTC. However, it does **not** wipe the registers. Instead, it changes the IP to point to the syscall handler in the protected shim.
- The syscall handler runs, interpreting the syscall. It performs some amount of emulation work inside the protected process, perhaps copying buffers to bounce buffers, or anything else that is required.
- The syscall handler informs the VMM that the next syscall is coming from the syscall handler, and that it should go to the guest OS. It also tells the VMM how many registers will contain parameters, and how many are expected to be overwritten by the guest OS kernel.
- The syscall handler performs a second syscall. The thread state is saved in a second buffer in the CTC. The VMM wipes all registers that do **not** contain arguments. The VMM changes the IP and SP to be the appropriate values for the trampoline.
- The VMM allows the kernel to run, and handle the syscall. The kernel may write to application pages (though, if things are working correctly, it will only write to unshadowed bounce buffers). It may also read data out of bounce buffers.
- The kernel, once it is finished with the syscall, returns to the protected process. However, what

[38] It is not clear how this information is propagated from the VMM to the shim.

it returns to is the trampoline code rather than the protected code. The trampoline runs, and asks the hypervisor to restore the thread state; the hypervisor validates the CTC and then restores thread state to the 2nd state saved (that which is **inside** the syscall handler).

- The syscall handler does any post-call cleanup, composes a return value, and then asks the VMM to return to the calling code.
- The code that originally performed the syscall runs.

5.9.4 Swap Out/Swap In

This section details how a page can be swapped out by the guest OS and then later swapped back in.

- While running kernel code, the guest OS may decide that it needs to swap out some pages. The pages that it chooses may include a private page from some application.
- The guest OS deletes page table entries from the appropriate page table(s) and then flushes the TLB on all processor(s). The VMM detects the page table modification, notices that a virtual page has been invalidated, and invalidates the related entries in the host page table.
- The guest OS attempts to write the page to disk. This, presumably, involves a DMA operation. DMA operations are treated as accesses from the system shadow; since the guest physical page in question is not currently mapped into the system shadow, this DMA requires that the VMM step in.
- The VMM encrypts the guest physical page, recording the (IV,H) value for the (RID,RPN). It then zaps the guest physical page and maps it into the system shadow.
- The DMA operation now proceeds, and the page is written to disk.
- The guest OS allocates the guest physical page for other purposes. Since it is mapped into the system shadow, the guest OS may read the page, write the page, or do anything to it.
- Later, the protected process hits a page fault on the virtual address. At this point, this becomes a normal read operation (see the example above); the guest OS reads the page from disk, the guest OS page table is set up, the access is retried and faults to the VMM, the VMM validates the guest physical page contents, and then sets up a host mapping.
  - The only difference here is that the VMM may still have an old cached value of the (RID,RPN) for this virtual address. Note that while the guest OS swapped the page out to disk, neither the shim nor the VMM noticed this. Both of them keep, in their tables, an entry that defines the (RID,RPN) for this virtual address.

5.9.5 mmap()

This section details the steps to `mmap()` a new virtual page.

- The shim determines the RID of the resource. If it is mapping a page from a file, then the RID is the RID of that file. If it is mapping new anonymous memory, then the RID is the RID of the object that represents anonymous memory.
- If the page that the shim is mapping is new (as in a new anonymous page), it allocates a new

RPN[39]. Otherwise, it determines the correct RPN of the existing page.

- The shim calls the kernel to `mmap()` the page. The kernel returns the virtual address allocated for this map.
- The shim adds a table entry, mapping this virtual address to the (RID,RPN).

At this point, the page is ready to use. The VMM, of course, does not know about the page, but (as detailed in the read example above), it will ask the shim for this information when it needs it.

5.10 Similarities

The two designs are very similar. Some noteworthy points include:

- Both designs include some mechanism, with higher authority than the guest OS kernel, which manages and enforces the various protections. Overshadow call this the “VMM,” while Bodyguard calls it the “hypervisor.”
- Overshadow's concept of an application is nearly identical to Bodyguard's concept of an entity.
- Both designs replace private pages with some sort of obscured version when code outside the protection boundary attempts to access a private page, thus preventing snooping.
- Both designs keep track of the working set of each protected process, and validate accesses, thus preventing corruption.
- Both designs use a shim to wrap a legacy process in nearly automatic protection.

5.11 Differences

There are also many noteworthy differences between the two designs:

- Overshadow encrypts private pages, while Bodyguard replaces them with signatures
- Overshadow encrypts pages, and thus, once a page is encrypted, they need store only the (IV,H) pair. Bodyguard instead replaces the page with a randomly-generated signature, and thus must store the entire private data buffer in hypervisor memory.
- Overshadow can discard old (IV,H) pairs when a page is modified. Bodyguard keep copies of all signatures (and their associated private pages) for the entire life of the entity.
- Overshadow uses a trampoline to tell the hypervisor when a protected process needs to run; Bodyguard can automatically detect this.
- When a protected process has called out to untrusted code, Overshadow stores register state in CTCs in virtual memory; Bodyguard stores this in hypervisor memory.
- Overshadow keeps track of private data as objects; virtual pages map to individual pages within objects. Bodyguard keeps track of each page individually, storing the contents as simple bytes.
- Overshadow's shim explicitly tracks all pages, and must eagerly keep the virtual page table up-to-date. Bodyguard implicitly discovers pages as they are used.

[39] It is not clear how the shim informs the hypervisor about new RPNs.

- Overshadow requires that the shim know the initial contents of all pages mapped into the process[40]. Bodyguard instead allows any initial contents at the moment of first access; if the shim expects certain contents, it must explicitly check for them.
- Overshadow keeps track of the virtual page table in the shim, and must feed this information into the VMM before it can be used. Bodyguard does this inside the hypervisor directly, with no shim interaction.
- Overshadow's shim includes both shadowed and unshadowed portions. Certain virtual addresses, then, can always be used for bounce buffers (in both directions) because those pages are always public, even when the protected process writes to them. Bodyguard, on the other hand, protects all pages in the process; any time that the process writes to any page, Bodyguard assumes that the data written is private data. This requires that Bodyguard's shim explicitly mark certain pages public (for outgoing bounce buffers) or remove them from our working set (for inbound bounce buffers).
- When a protected process `fork()`s, Overshadow marks the page as COW, and splits it into independent logical pages if/when it is written to. Bodyguard explicitly splits all of the logical pages immediately, at the moment when `fork()` occurs.
- Overshadow does not have a safe bootstrap strategy. Bodyguard does.

5.12 Tradeoffs

Encryption vs. Signatures

The largest difference between the two designs is the question of how to obscure private pages when they are touched by untrusted code. Overshadow encrypts the pages, and saves only the initialization vector and the hash; Bodyguard replace them with a signature.

The most obvious tradeoff is, of course, CPU time vs. memory. Bodyguard consumes memory, which cannot be recovered until the entity dies (although, of course, we could imagine swapping old signatures out to disk). Overshadow, on the other hand, consumes a significant amount of CPU every time that the VMM must either encrypt or decrypt a page.

However, there is a more subtle tradeoff: that of explicit vs. implicit protection of the working set. To understand the advantage that Bodyguard has, consider the process of opening a file stored on the guest OS disk. The protected process opens up the file, and maps one or more pages into new virtual addresses. For the moment, assume that this file was written to the disk by this entity at some earlier time, and that the guest OS is working correctly (loading the correct pages).

In Overshadow, the shim must explicitly track exactly what file this is, and map it to a protected object. The VMM must load up all of the metadata for this object, and each new page must be explicitly associated with an (RID,RPN) pair. In most cases, this is straightforward, but there could be complexities. For instance, if a file was hardlinked or copied from one location to another, could the shim automatically detect the correct resource for this page?[41] Moreover, the process of loading the

[40] This author is not sure if this requirement applies to unshadowed pages in the protected process. But certainly, it applies to all shadowed pages.

[41] One could imagine including the RID in the file header of each file. However, in order for this to work, the first page of the file header would have to be public data, so that the process could map the page (into the unshadowed portion of the shim), read the RID, and then later map the other pages of the file with the correct (RID,RPN) values.

metadata from disk into the VMM seems an undesirable layer of complexity.

In Bodyguard, the shim doesn't need to know about exactly which file it is opening; it can simply open a file and map it. The newly-mapped page may contain a signature, and the hypervisor will automatically detect this and give the process access to the private data. In this design, a file can be self-validating; the file header can contain a magic value in the private contents of the first page, and then have a system of checksums to validate the size and contents of the rest of the file.

This system cannot, taken in isolation, validate that the contents of a file are up-to-date and correct (an attacker could present an out-of-date copy of the file, or the wrong file), but it can validate that the contents of the file are the correct and complete contents of **some** file written by this process at some time. When it is necessary to make sure that the file contents are both correct and up-to-date, it can simply store a secret value in the file header, and validate this secret when you open up a new copy of the file. (This is equivalent, however, to storing the known RID for some file and thus costs us some of our advantage.)

In either system, it is possible to map an arbitrary, unknown file, figure out how to map the rest of the file, and validate its contents. However, this is difficult in Overshadow, as Overshadow requires that the VMM know the (RID,RPN) of every private virtual page, so that it can look up the proper (IV,H) values to validate the pages. Our design doesn't need to know this ahead of time; a protected process can read a signature, map it to a private value, and thus implicitly determine the proper private value for this page.

The encryption/signature tradeoff, then, is actually two tradeoffs. First, we can trade off CPU time vs. memory consumption. Second, we can trade off explicit vs. implicit mapping of pages.

Explicit vs. Implicit Virtual Space Management

The previous section has already discussed some of the explicit/implicit tradeoff, focusing on its implications for loading files from disk. However, the question of explicit vs. implicit virtual space management is more general. The basic question is: “when the protected process `mmap()`s a new virtual page in the guest OS, does it need to inform the VMM?”

Overshadow uses an explicit approach. The shim must immediately record the (RID,RPN) for this virtual address; the VMM will not know about it immediately, but it will draw this new entry into the cache as soon as it is necessary. This design means that the shim defines the required contents of all such new pages, before it is able to touch them.

Bodyguard uses an implicit approach. The shim simply `mmap()`s the page, and the hypervisor automatically adds a new virtual page to the working set the first time that it gets used. The hypervisor does not know the contents of the page until the first access, and the untrusted OS is free to fill the page with any values, whether they be OS-generated values, zeroes, or a signature of a private page from long ago.

At first glance, the implicit design seems more simple, and thus attractive; the shim doesn't need to perform as much manual bookkeeping when pages are mapped. However, in practice, this is not truly the case; many newly-`mmap()`ed pages are required to have precisely the values the protected process expects, whether it be all zeroes for new MAP_ANONYMOUS pages, or the correct signature for

It is worthwhile to note, however, that the Overshadow paper does not discuss anything like this possible design. It seems that Overshadow, as described in the paper, is unable to deduce the RID of a newly-opened file; the shim must simply know this beforehand.

newly mapped private pages. This means that the shim still has to some bookkeeping to do; in this case, it has to read the contents of most pages immediately after they are `mmap()`ed, and validate that they are correct. That is, the shim `mmap()`s the page, and then immediately begins to read it; the hypervisor imports the current contents, and assumes that they are correct, but the shim must validate the page before returning. Thus, the apparent simplicity of implicit mappings vanishes in many cases.

Thus, we see that, for many pages, the explicit and implicit mechanisms are equivalent; we need to perform some sort of bookkeeping, right at the moment of `mmap()` – and before any other use of the page – which allows us to ensure that the contents that the guest OS gives us are correct.

As discussed in the previous section, we have two different types of maps that a protected process might need to perform. In one case, the maps are expected to have very precise contents (the contents of a file containing private data, for instance); in this scenario, the explicit methodology makes the most sense. In another case, the maps are data imported from the guest OS, which have contents that the protected process may not be able to predict; in this scenario the implicit methodology makes the most sense. We can implement either type of operation in either an explicit or implicit system, but each is easier in one system and harder in the other.

An alternate solution, which we hope to explore in our Future Work, is to augment the implicit methodology with the ability to register hashes for newly `mmap()`ed pages. That is, if the shim happens to know that a newly `mmap()`ed page must have a certain contents, it can communicate the hash of those contents to the hypervisor before it actually touches the page. The hypervisor will create a new virtual page object for that address, associate it with a new logical page object, and save the hash value as the "current contents." When the page is first used, the hash will be verified; the hash will then be discarded, and the normal verification process will be used from then on. We believe that this hybrid methodology might combine the best aspects of both designs.

Object vs. Page Protection

Both Overshadow and our work protect the contents of pages. However, while Bodyguard views pages as independent of each other, Overshadow views each page as a member of some object.

The tradeoff between the two designs is hypervisor complexity vs. shim complexity. Their design has more inherent complexity, since the hypervisor must manage objects, whether or not the shim needs this feature. However, in our design, when object-based protection is required, the shim must implement this inside the protected process, which adds complexity to the shim and consumes memory inside the (often limited) virtual address space.

We believe that Bodyguard's design is superior; we prefer to have our complexity in the shim, rather than in the hypervisor. One of our design points was to make the hypervisor as simple as possible, so as to minimize the likelihood of vulnerabilities within it.

CTCs vs. Hypervisor Protection of Registers vs. Incoming Handler

Overshadow protects registers from corruption using CTCs. CTCs are buffers in the virtual address space of a protected process that store the saved register contents when a thread is interrupted or otherwise jumps into the kernel. The job of the hypervisor is simply to verify that an address, given by the trampoline, contains a valid CTC, and then to restore the register state.

In our design, the hypervisor keeps the register state in private hypervisor memory. This allows us to detect when the code returns to a protected process, and to verify the registers, without needing CTCs stored in virtual memory.

The Overshadow design is simpler, and requires less hypervisor memory. However, it lacks the ability to automatically detect when code returns to a protected process, which is a desirable feature.

We suggest that the ideal solution would be a hybrid. In Future Work, we discuss the possibility of an "incoming handler." This is a handler, analogous to the syscall handler, which is called any time that the untrusted OS attempts to return to a protected process. The hypervisor automatically detects the jump into protected code, and forces the code to the incoming handler instead. The shim then verifies the registers, without any need for further hypervisor action. In this design, the the hypervisor is responsible for writing thread state to the CTC when the process jumps out of protected code, but does not perform any validation when code jumps back. Instead, this is handled entirely within the shim.

Chapter 6: Prototype Implementation

This chapter details the implementation of a Bodyguard prototype.

It starts with a general overview, and then discusses the codebase we modified. It describes how the code intercept TLB fill operations in order to implement Bodyguard's protections.

It then discusses each of the major structures used by the code, explaining how the major fields are used. It also details the design of the major functions that run inside the hypervisor.

Next, it describes how the current implementation of the shim operates.

It closes with a TODO list (points that need to be fixed in the current implementation in order for it to match the design defined in Chapter 4 above), and a list of points of interest that we believe would be interesting to other implementers.

6.1 Overview

Our implementation is based on the Bochs x86 emulator [28]. Inside that, we run an unmodified copy of Linux (RedHat 6.2). We are able to protect unmodified versions of most common Linux tools.

We have implemented a shim that automatically provides memory protection and that emulates most syscalls. Protected applications have complete memory protection, once the shim is initialized; the hypervisor uses signatures, as described in Chapter 4, to protect private data from both corruption and snooping.

The internal hypervisor logic is implemented in a well-isolated module that interacts with Bochs through a well-defined interface. The intent of this implementation is to make it possible to quickly port this to other VMMs or emulators.

Much work remains to be done, however. The hypervisor currently can only handle one entity at a time. It currently only implements memory protection; register protection has not yet been implemented. Even its memory protection is limited; it does not protect against device accesses, and cannot protect against attacks while the processor is in real mode. While the hypervisor already implements the virtual page/logical page distinction, shared logical pages have not yet been implemented. `fork()` has not been implemented. Moreover, the shim is currently implemented as a shared library, which is loaded into the protected process using the LD_PRELOAD environment variable; the secure initialization process has not yet been implemented. In addition, there are a variety of small hacks and assumptions scattered throughout the code that must be fixed and generalized.

Additionally, in places where it is impossible for the shim to perfectly automatically protect (such as knowing whether a file should be written to the untrusted OS as private or as private pages), the shim makes the practical assumption to write out the private data. This works well for now, since we are generally wrapping existing Linux tools, which must interact with the outside world, writing out buffers that do not need protection. However, future versions of the shim may reverse these assumptions. The shim does not yet have the ability for the protected process to call into it and define which files should be private and which should be public.

Finally, because of the inherent limitations of Bochs (slow speed, missing features), the current implementation is not really ready for use as an ordinary operating environment. Our hope is that, once we port this to Xen, the performance will be good enough that we can deploy and use these features in a general-purpose, interactive Linux box.

6.2 Modifications Made to Bochs

Our prototype is based on Bochs (a recent version checked out from CVS). It was configured using the following command:

```
./configure --enable-clgd54xx
```

In addition, we turn on Bochs assertions by editing `config.h` and setting `BX_ASSERT_ENABLE` to 1.

We have made the following modifications files to Bochs files:

- `Makefile`: build the hypervisor/ directory (see details of our code below)
- `memory/misc_mem.cc`: report physical memory size to the hypervisor in init_memory().
- `cpu/cpu.h`: add (perhaps redundant???) `execOK` and `writeOK` flags to the `bx_TLB_entry` struct, so that we can mask off certain types of access in TLB entries.
- `cpu/exception.cc`: notify the hypervisor in `interrupt().` The hypervisor may tell Bochs to flush the TLB (which happens if we were running a protected process).
- `cpu/paging.cc`: add code to treat a TLB entry as invalid if the access is masked off (see above). Then add code to call the hypervisor to validate all types of access (see details of the hypervisor code below). The hypervisor may tell Bochs to flush all old TLB entries before putting this entry into the TLB. It may also tell Bochs to mask off certain types of access in this entry.
- `cpu/soft_int.cc`: add code to intercept system call 400 (`int 0x80, EAX=400`) and send it to the hypervisor as a hypercall. Add code to notify the hypervisor on all other system calls (`int 0x80`), allowing the hypervisor to redirect the system call to the syscall handler if appropriate. (The hypervisor decides whether to redirect or not; code in this file actually performs the redirect, if necessary.) Implement in this file (without notifying the hypervisor) hypercall 14, which returns from the syscall handler to the calling program.[42]

New Files for Added to Bochs

We added four new files to the Bochs source:

- `hypervisor.h`: declares the interface between Bochs and the hypervisor
- `hypervisor/Makefile`: copied from some other Bochs directory, this just builds the hypervisor code in the format that the main Makefile expects.
- `hypervisor/core.cc`: implements the hypervisor logic

[42] Hypercall 14 is a hack, and we hope to remove it in Future Work. It exists only because the author did not have time to investigate exactly how to create a correct stack frame. Instead, the syscall intercept code in this file will push many registers on to the stack in a non-standard format, and hypercall 14 will pop those registers from the stack again.

- hypervisor/bit_key_tree.h: declares a custom data structure, used by the working set code. This will be detailed later.

6.3 TLB Intercept

As described in previous chapters, Bodyguard needs to mask off certain page table entries (or, certain types of permissions from certain pages) in order to implement its protections. For instance, when running untrusted code, the CPU should never be able to access frames that contain private data, and should never be able to write to pages that have logical pages mapped.

To implement this, our code intercepts all TLB fill operations within Bochs. In Bochs, all accesses to virtual memory are funneled through a TLB mechanism, which is an array of entries (analogous to a hardware TLB) that represent a cache of page table entries. When code attempts to execute, read, or write a certain virtual page, Bochs checks the TLB for an appropriate entry; if the entry does not exist, or if it does not have the correct protection bits, it parses the page tables, looking for an appropriate entry. If it finds such an entry, it updates the TLB; if it does not, a page fault occurs.

We added two flags, execOK and writeOK, to the TLB entry struct. These flags allowed us to create a TLB entry that explicitly masked off those types of access (there are no circumstances where Bodyguard needs to mask off read access while allowing other types of access, so we didn't define a flag for that)[43]. We modified Bochs to check these flags on all accesses. For instance, if code was attempting to execute a given virtual page, but execOK was FALSE, Bochs rejected the access (causing a new page table lookup).

When Bochs has to look up a virtual page in the page table and finds an appropriate entry (that is, a valid entry with access bits that match the access being attempted), we intercept the code before it actually adds the TLB entry to the table. Bochs calls execPage(), readPage(), or writePage(), depending on the type of access. Our code in the hypervisor takes this opportunity to update the frame state (if required), to do logical page bookkeeping, and to detect jumps into or out of the protected process.

In situations where execPage() detects that the processor is jumping into or out of a protected process, execPage() will return FALSE to Bochs. In this case, Bochs will clear the TLB before adding in the new entry. We thus maintain an invariant: at all times, the TLB contains only entries that are valid for the current running entity. If we are running a protected process, all of the TLB entries reflect page accesses that are valid for this protected process. Pages that have signatures showing are not present in the TLB at all; pages that contain public data might be present in the TLB, but execute permission is masked off.

Thus, any time that the CPU jumps into or out of a protected process, there is a period of time where the hypervisor is called for each access, as the TLB is filled. However, once the TLB is filled, the hypervisor is not called any more. We considered using a “tagged” TLB, where entries would include markers to indicate whether they were valid from protected code, or from untrusted code, but have not yet implemented that enhancement.

[43] It is possible that these flags are redundant; we suspect that we could accomplish the same with existing fields in the Bochs TLB entry struct. However, the Bochs implementation, as it models the x86 hardware rather closely, was difficult to understand. We chose the expedient hack of adding new flags.

6.4 Structures

6.4.1 Entity

This represents an entity. It contains linked list of processes[44], and one pointer to a "pendingProc," which is a pointer to a Process object that was created in a recent `fork()`, but which has not yet run. The pendingProc is added to the entity's process list as soon as the hypervisor can determine its process ID.[45]

The entity also has a hash table of saved pages. To look up a saved page, the hypervisor slices out 32 bits of the signature from a certain location, then uses it to index into a BitKeyTree (defined below). The result is a (perhaps empty) linked list of pages with that hash.

The entity also has a list of Shared pages (currently unused, will probably change to a BitKeyTree when we implement shared logical pages).

Finally, the struct has a flag that indicates whether or not the entity is fully initialized. This is necessary because the current shim is a shared library, rather than our more correct bootstrap mechanism; complexities arise if you have private and public pages calling each other inside the same shim initialization routines.

6.4.2 Process

This represents a single process in an entity. It contains all of the expected fields, including its ID, a BitKeyTree for the working set, and information about the syscall handler.

In addition, it includes fields that support the current shared-library based shim implementation. `pendingAdd` and `pendingAddLen` give the virtual address and size of the next range of memory that the shim intends to add to the process. `implicitWorkingSetExpansionEnabled` is a flag that indicates whether the hypervisor should automatically add new pages to the working set or not. The operation of all three of these fields will be detailed in Section 6.6 below.

6.4.3 VirtualPage

This represents a single virtual page in the working set of a protected process. It gives the virtual address, and a pointer to the `LogicalPage`.

6.4.4 LogicalPage

This represents a single logical page in an entity. It includes fields that indicate whether or not it is shared, and if so, how many virtual pages map it. It contains a flag that indicates if the page is currently public or private.

Most importantly, it contains a pointer to a `Frame` and a `SavedPage` struct. At all times, at least one of these fields must be non-NULL. If both are non-NULL, then this means that the frame contains the current contents of this page (or its signature), and that the contents are read-only (that is, they have not

[44] Our current implementation only supports one process per entity, since `fork()` has not yet been implemented. But adding support for multiple processes is on the TODO list.

[45] We have started implementing `fork()`, but it doesn't work yet. This is part of the current prototype implementation.

been modified since the last signature was taken). If only `frame` is non-NULL, then this means that the frame contains the contents of the page (definitely not a signature). If only `savedPage` is non-NULL, then this means that the page is not known to be currently present in any frame, and thus the contents are saved in the `SavedPage` struct.

6.4.5 SavedPage

This represents the saved contents of a public or private page.

The primary fields are `front`, `back`, and `contents`. `contents` is a pointer to a page-sized buffer of data that is a duplicate of the public contents of this page (what the hypervisor expect the OS to put into a frame). If the page is public, then this will be the contents of the page. If the page is public, then this will point to the signature. This simplifies the code when the hypervisor is doing page content validations, as we can simply compare the frame contents to the buffer pointed to by the contents pointer.

`front` and `back` are offsets into a temporary file written by the hypervisor to its private disk. Each time that the hypervisor creates a new signature, the current private contents and the signature generated are both written to the file, and the offsets to those pages are stored in the `SavedPage` struct that is created to represent them. This has two purposes. First, it reduces memory consumption, since the hypervisor doesn't need to address two pages worth of data for each signature generated[46]. Second, it allows the hypervisor to indicate when two frames happen to contain the same contents, and potentially to share them. That is, when the hypervisor finds that the OS has written a signature to some page and that it must show the private data (or vice-versa), it doesn't actually read the page up from disk; it `mmap()`s the correct file offset to the correct buffer location. This makes it possible for multiple frames to share the same backing file, if necessary.[47]

`SavedPage` also includes a pointer to another `SavedPage` called `alias`. When valid, this means that this struct is an alias of another; this happens when the hypervisor finds that it must merge two `SavedPage` structs into a single struct, but it is unable (at the time of merge) to determine all of the `LogicalPage` structs that would need to be edited. Later, when some `LogicalPage` uses this struct, it will find the alias; it will edit the `LogicalPage` to point to the other `SavedPage`, and (once all such references are cleaned up) this `SavedPage` will be freed.

(Note that when alias is non-NULL, all other fields in this struct are invalid.)

6.4.6 Frame

The hypervisor has an array of Frame structures, initialized when Bochs calls us to tell us how many physical pages it will use. Each struct represents one frame. It includes a pointer to the physical memory (inside Bochs) that stores this frame. It also includes a pointer to the `LogicalPage`[48]

[46] Of course, we still consume one page of virtual memory, because we always have a mapped copy which is pointed to by `contents`. In the future, we might choose to discard even that copy, so as to save memory.

[47] This is another thing we need to think out more clearly in the future. Are there compelling cases where pages might have identical contents but the untrusted OS wouldn't know that? If two pages are COW copies of each other, we expect the untrusted OS to have them share a frame...and if the guest OS splits them, then we expect that one is about to get modified anyhow. Perhaps this is working too hard to solve a simple problem?

[48] Our design calls for the ability to map multiple logical pages into the same frame, but the current implementation hasn't implemented this. If multiple logical pages use the same frame, then we will swap the mappings back and forth based

mapped into this `Frame`, if any.

This struct includes two flags that indicate the signature state. `sigShowing` applies only when a private logical page is mapped into the frame; it tells us whether the frame currently contains the private data for this logical page, or its signature. `sigScanNeeded` is set any time that any process modifies the frame, except when a LogicalPage is mapped to it. It is cleared when the hypervisor scans the page to figure out whether or not the recent modifications have written a signature to the frame[49]. The idea of this latter flag is to eliminate useless scans for signatures, particularly in `execPage()`.

6.4.7 SharedPage

This structure is currently unused, because shared logical page is not yet implemented. Currently, it functions as a list node in our list of shared pages for each entity; in the future, we will probably change this to use a BitKeyTree instead, meaning that we can delete this struct.

6.4.8 BitKeyTree

BitKeyTree (BKT) is a new data structure we developed for this project. It is, essentially, an n-ary tree, with the special property that, given a fixed-size key, it can index to any entry in the tree (or, find that a key does not exist) without any conditional branches. In this code, the BKT is implemented as a template.

Each node in the BKT tree has a fixed power-of-2 number of child nodes. Each node represents some subset of the space of possible keys, and each child represents equal portions of that space. Thus, we don't have to compare our key to a key stored in each node; instead, we slice bits out of our key, and use them to give us an index into the array of child pointers. That is, if we have 16 children in each node, then at each level of the tree, we slice the next 4 bits out of our key, and use it to choose which child to go into.

Of course, a key might not exist in the tree. Rather than using NULL pointers to represent empty subtrees, as might be common, the BKT represents an empty subtree with a “dead” node where all of its child pointers loop back to itself. Thus, we are guaranteed, for any key, to be able to follow the fixed number of pointers through the tree...although we might get into a loop.

Once you have followed a fixed number of pointers, you then inspect the node to see if it is alive (and thus the thing you are looking for) or dead.

The point of a BKT is that it should be much faster than an ordinary binary tree, since failed branch prediction is so costly in modern processors. This code never makes a branch (until we are checking to see if the node is dead or alive) and never has to read values from the tree (other than child pointers). Since each BKT has a fixed key width, the lookup algorithm needs to make a fixed number of steps; we thus expect the compiler to unroll the loop. The result should be very dense code, with the entire lookup being accomplished (in some architectures) with as few as two instructions per step.[50]

on usage.

[49] Note that we don't have any way to store that a frame matches a saved page, except to associate it with a logical page. Thus, we only perform this scan, and potentially clear the flag, when we are performing some sort of access where we might be able to associate a logical page with the frame. If the signature that we find which matches is not from the right entity, then we will not make the association, and we will not clear this flag.

[50] Note that we have not yet had the opportunity to verify that this new data structure is actually faster. However, this paragraph describes the design intent.

6.5 Functions

6.5.1 untrustedOS_execPage()

Summary

This function is called by Bochs when Bochs needs to add a TLB entry for "execute" access. That is, the code is attempting to run code at a certain virtual address, and the TLB does not yet have an entry for this page (or, the entry has exec permissions masked off). However, Bochs searched the page tables and found a page table entry with execute permissions for this virtual address. Before it adds this entry to the TLB, Bochs calls this function in the hypervisor.

The purpose of this function is to detect jumps to and from protected processes. It must determine whether or not the frame pointed to by this virtual address contains a private page. Also, if it contains a private page, then this function will make sure that the private data is showing (not the signature). (Note that if the frame contains a page that is in the working set of some process, but it is public in that working set, then we treat this as running untrusted code. Only private pages are treated as running trusted code.)

Flushing the TLB

The hypervisor must determine whether this call jumped into or out of any protected process. To do this, it keeps track of a `curProc` variable, which points to the process that is currently executing. (If untrusted code is running, then this is NULL.) The last task of `execPage()`, just before it returns, is to update this pointer.

The return value from this `execPage()` is `true` if the hypervisor wants Bochs to flush the old TLB entries before putting this new one in; it returns `false` if the TLB entries may remain. Effectively, this means that `execPage()` returns `true` iff `curProc` was altered by this call.

Mask

`execPage()` is responsible for delivering a mask value to Bochs, which indicates which types of permissions the TLB is allowed to have. Currently, it either returns `READ|EXEC` (mask off write access) or `EXCEPTION` (described below, means that Bochs should force a page fault).

First Page in the Working Set

`execPage()` also has the responsibility, with the current design of the shim, of adding the very first page to the working set of the first process. (See Section 6.6: Shim Implementation below.)

Validation

Finally, if the code page being executed is private, then `execPage()` figures out which process is running and performs validation of the code page against the expected value in the working set. If there is any problem with that process, then `execPage()` will kill the entity and then tell Bochs that corruption has occurred. Bochs will then cause a page fault to happen, rather than let the code execute.

Note that, while we have not yet implemented register protections, the register verify/restore code would go in `execPage()` (as the CPU jumps back in). The register save/wipe code, however, will have to go in `startInterrupt()` (see below).

6.5.2 untrustedOS_readPage()

This function is called by Bochs when code is reading a value from memory, but the TLB doesn't have an entry for this page.

If `curProc` is NULL, then the hypervisor knows that untrusted code is running. In this case, things are fairly simple; it just checks to see if the frame has a logical page mapped, if the page is private, and if the private contents are showing. If so, then it generates a signature (if the page didn't already have one) and the replaces the private contents with the signature.

When `curProc` is not NULL, things are more complex. First, if the page already exists in the working set, then `readPage()` will validate the contents of the frame. Otherwise, it may import the page into the working set of the process.

Also, when `curProc` is not NULL, the hypervisor has to check to see if the frame contains a signature, and restore it to its private data.

As with `execPage()`, `readPage()` returns a boolean, telling Bochs whether a TLB flush is required or not. (This is only because of a very strange case in FindInWorkingSet, a function called by `readPage()`. This case will go away when we implement the secure bootstrap methodology.)

Finally, like `execPage()`, `readPage()` returns a mask. In this case, the only possible masks are `READ` (mask off execute and write) and `EXCEPTION` (corruption detected, force a page fault). Note that it is possible that the page might be executable, or writable...but if so, then Bochs will run those appropriate functions if and when the program tries that type of access.

6.5.3 untrustedOS_writePage()

This function is called by Bochs when code is writing a value to memory, but the TLB doesn't have an entry for this page, or write access is masked off in that entry.

The first step in `writePage()` is to call `readPage()`. This performs all of the normal `readPage()` work, including adding the page to the working set (if required) and content validation (if required). If `readPage()` returns a mask of `EXCEPTION`, then `writePage()` immediately returns that to Bochs. However, if `readPage()` returns a mask of `READ`, then `writePage()` will do some work to make sure that the page is ready for writing. It will eventually update the mask to `READ|WRITE`, and then return to Bochs.

`writePage()` has to following tasks, in addition to what `readPage()` did:

- If the CPU is running untrusted code but the frame has a logical page associated, unmap the logical page from the frame, making sure that it has a `SavedPage` struct first.
- If the CPU is running untrusted code but no logical page is associated, set `sigScanNeeded` on the frame.
- (Later, when we implement the ability to map multiple logical pages to a single frame): If the CPU is running trusted code but the frame has two or more logical pages associated, unmap all of the logical pages except for the one which is involved in the write.
- If the CPU is running a protected process, discard any `SavedPage` from the logical page associated with the frame.

– If the CPU is running a protected process, and the logical page is currently marked public, then mark it as a private page.

6.5.4 FindInWorkingSet()

FindInWorkingSet() is a major helper function, called by several other functions. Its primary purpose is to search the working set of the current process for a given virtual page. However, it also performs implicit adding of pages to the working set. It returns the virtual page that it finds (or creates)[51].

In addition, it has a boolean (inout parameter) that it returns, which indicates whether this function needs Bochs to flush the TLB.

6.5.5 untrustedOS_interrupt_400_handler()

This function is called by Bochs any time that any code executes int 0x80 with EAX=400. It is used to implement hypercalls. Bochs passes the registers EBX, ECX, EDX, ESI, and EDI as 5 parameters. (These are the same registers that Linux uses for its syscall parameters, making it easy to write a shim that implements hypercalls.)

Note that while we expect most hypercalls to be performed from inside protected processes, some of them (such as “create entity”) can be called from outside one. Each hypercall implements whatever assertions are appropriate for it.

There are currently 15 hypercalls; each one is fairly straightforward, and documented in comments in the code.

Like execPage(), this function returns a bool. If it is true, then Bochs will flush the TLB.

NOTE: Hypercall 14 (return from syscall handler) is implemented in the Bochs code, and does not call this function. See Section 6.2: Modifications Made to Bochs above.

6.5.6 untrustedOS_syscallAlert()

This function is called by Bochs any time that any code executes int 0x80 with EAX!=400. This notifies the hypervisor that someone is attempting a syscall.

If curProc is NULL, it returns 0 immediately. Bochs then handles the syscall using normal mechanisms (sending it to the untrusted OS).

If the syscall handler is defined, and if the protected process has **not** requested that the next syscall be allowed to go through to the kernel, then syscallAlert() will return the address of the syscall handler to Bochs. Bochs will push a non-standard[52] frame onto the stack (saving stack variables, and pushing arguments), and then force code to jump to the syscall handler.

[51] During initialization of the shim, it can return NULL, if the page does not exist and does not match the correct conditions for adding it to the working set. This case will vanish, however, when we implement the secure bootstrap process.

[52] Our intent is to pass a standard stack frame at this point, so that the syscall handler can simply perform a function return later, when the handler is complete. However, the author did not have time to investigate the exact format for a stack frame, so instead Bochs pushes a non-standard frame, and hypercall 14 is used return from the syscall handler.

If no syscall handler is defined, or if the protected process requested that the next syscall be allowed to proceed, then `syscallAlert()` will return 0, and Bochs will handle the syscall in its normal mechanism (sending it to the kernel).

Note that this function also has prototype code for `fork()`, although it is not yet operational.

6.5.7 untrustedOS_startInterrupt()

This function is called by Bochs any time that any sort of interrupt or exception occurs. Its primary purpose is to set `curProc` to NULL. It also tells Bochs whether it is necessary to flush the TLB or not.

Later, when register protections are implemented, this is where the hypervisor will save and then wipe the registers.

6.6 Shim Implementation

6.6.1 Loading the Shim

The current shim is implemented as a shared library, which is linked into an existing application using the `LD_PRELOAD` environment variable. We run a process with a command such as:

```
LD_PRELOAD=shim2.so[53] ls -al
```

`LD_PRELOAD` instructs the loader to load the library even if it is not in the dependency list of the application; its init function runs after all of the various pages are mapped, but before any other initializers.

This process, obviously, is not secure, since an attacker could corrupt the shim before it initializes itself. We plan to change to the secure bootstrap process in the future. However, this chapter will describe the shim as it currently functions.

6.6.2 Shim Initialization

The shim init function is called automatically, by the dynamic linker, before any other initializers. Its job is to get all of the pages into the correct state so that it can protect the application, which will run later.

To do this, the shim first opens up `/proc/self/maps`, and parses it to get a list of all of the maps in the process. It also opens up a temporary file, and immediately unlinks it (so that it will get cleaned up when the process dies).

For each range of pages in the process (except stack[54]), it writes the entire range out to the temporary file, and then remaps it[55]. It then calls the hypervisor, asking it to add the range of pages to the process.

[53] For legacy reasons, our shim is called shim2.so. A previous version of the shim actually used two libraries; now, only one is required. But we haven't gotten around to changing all the references to shim2.

[54] The process we're about to perform would corrupt the current stack frame, since it would effectively take that one page back in time a few milliseconds. Instead, we'll let the process just discover the stack pages implicitly, as they get touched.

[55] It calls `mmap()` with the flag MAP_FIXED.

(This sets the pendingAdd and pendingAddLen fields in the Process struct.) Next, it touches each page in the range. Each touch results in Bochs calling readPage(), which calls FindInWorkingSet(). FindInWorkingSet() will notice that the page address matches pendingAdd, and will add the page to the working set. (This will be detailed just below.)

Once all of the pages are added to the shim, it calls two hypercalls, "mark entity initialized" and "turn on implicit working set expansion", which tell the hypervisor is ready for normal running. At this point, the normal protection mechanisms kick in.

Finally, the shim registers the syscall handler, and then returns. The normal code of the process then runs, performing init on other libraries (if required) and then eventually calling main().

6.6.3 Adding Pages to the Working Set

When FindInWorkingSet() adds a page to the working set because of pendingAdd, it will mark this page as a private page even though the contents are created by the OS. (Normally, with implicit page adds, such a page would be viewed as a public page. This hack is necessary because the hypervisor must be able to detect that the CPU is running private code, even though the current reality is that the untrusted OS loaded the page contents.)

However, there is an added complexity, in that the contents of these pages are known to the untrusted OS, and it believes that the pages are clean. Thus, he might legitimately discard the pages from memory (to free up space for others) and then later read them back from disk. He expects that this will work just fine. Thus, we have two options: either notify the untrusted OS that the pages are dirty (so that it will read them again, and see the signature, and flush that to disk before discarding them), or else tolerate that it might later move a virtual page to a new frame, and initialize that frame with the private data. We chose the latter. When FindInWorkingSet() adds a page because of pendingAdd, it immediately sets up a SavedPage for it. However, this SavedPage is unlike any other: although it is theoretically a SavedPage for a private page, the front and back (that is, the private data, and the signature) are identical! Thus, if the untrusted OS discards the page, and later reads it from disk again, the contents that it reads will match the false "signature" that FindInWorkingSet() created. The hypervisor then show the private data (which will have no effect).

6.6.4 Syscall Handler

For convenience, the syscall handler is actually divided into two parts. The function that is registered with the hypervisor is called syscall_handler(). This function does a little bit of logging, and then calls into the second part, syscall_handler2(). When syscall_handler2() returns, syscall_handler() does more logging, and then uses the hypercall to return to the regular code. The point of this division is to make it easier to write handlers; they can say "return X;" rather than deal with the complexities of logging and hypercalls.

syscall_handler2() is implemented as a massive switch() statement. (The default: case prints out a warning about an unimplemented syscall, and then crashes the application.)

Many syscalls are trivial to handle. If the syscall does not have any pointer arguments, then the call can usually be passed directly to the kernel with no emulation. A few, such as close, will require emulation when we implement private file support inside the shim, but are trivial for now. A few others, such as `fork`, also require some amount of emulation.

The majority of the code in `syscall_handler2()`, however, deals with bounce buffers. Bounce buffers are required for any pointer that is sent to the kernel. To handle this, the function declares two buffers, each page-aligned and of size `BOUNCE_BUF_LEN` (currently 64 pages, that is, 256 K). These buffers are declared as stack variables, meaning that they eat up a large amount of stack space, but this doesn't effect performance much unless the shim has to wipe them.

The function also declares 3 macros that are used by the various syscall implementations:

- `BUF1_OUT()`: Marks all of buf1 to be public pages, so that the kernel can read the contents.
- `BUF2_IN()`: Discards all of buf2 from the working set, so that the kernel can overwrite them. However, the hypervisor will make sure that all of the pages are showing their signatures before they are deleted, so that no private memory will leak.
- `BUF2_INOUT()`: Marks all of buf1 to be public pages, and then deletes them from the working set. Used when the same pointer sends data to the kernel but then also will be used to read data back. The kernel will see the private contents, but can also overwrite the buffer.

6.6.4.1 Example syscall wrapper: read

Note: arg1, arg2, arg3 are the syscall arguments.

arg1=file descriptor

arg2=buffer pointer

arg3=length

```
case 3: /* read */
  assert(arg3 < BOUNCE_BUF_LEN);
  BUF2_IN();

  {
    int rc;
    AllowNextSyscall();
    rc = syscall(3, arg1, buf2, arg3);

    if(rc > 0)
    {
      assert(rc <= arg3);
      memcpy((void*)arg2, buf2, rc);
    }
```

```
        return rc;
    }
```

6.6.4.2 Example syscall wrapper: write

```
  case 4: /* write */
    assert(arg3 < BOUNCE_BUF_LEN);
    memcpy(buf1, (void*)arg2, arg3);
    BUF1_OUT();

    AllowNextSyscall();
    return syscall(4, arg1, buf1, arg3);
```

6.7 TODO List

This section is a list of items that must be fixed in order for the implementation to match Chapter 4: Bodyguard Design. Features that are listed in Section 8.1 (Future Work) will not be discussed here.

- Implement secure bootstrap
- Implement register protection
- Implement `fork()`
- Implement shared memory
- Allow more than one logical page to be mapped into the same frame at the same time
- Implement support for the rest of the syscalls in the shim
- Implement private IPC
- Implement private files (with content validation)
- Implement functions, which can be called by a modified application, to specify what sort of protections are required for files, IPC, etc.
- Implement support for multiple entities
- Remove debugging code in hypervisor & shim
- Remove asserts in the hypervisor that depend on the shim working correctly. Crash a badly-behaving entity, rather than crashing Bochs.

6.8 Difficulties and Lessons Learned

This section contains discussion of various difficult problems, discoveries, and random points of interest that were encountered during implementation of this work.

Accesses During an Interrupt

Our initial design said that we would catch transitions to the kernel by watching `execPage()` for attempts to execute untrusted code. We would set `curProc` to NULL when that happened. However, what we found instead was that Bochs was accessing[56] a few pages after an interrupt or exception happened, but before any kernel code ran. We didn't dig deeply into the problem, but we guess that was reading the exception vectors to find out where the code should jump.

We solved this by dropping a special call into the `interrupt()` function in Bochs; it calls `untrustedOS_startInterrupt()`. The hypervisor sets `curProc` to NULL, tells Bochs to flush the TLB if `curProc` had been non-NULL, and (will in the future) save and wipe registers.

Finding Syscall Specifications

It's difficult to find specifications for Linux syscalls. The man pages are often helpful, but sometimes information is sketchy. We found an excellent page at http://bluemaster.iu.hio.no/edu/ca/lin-asm/syscalls.html, but this page is offline at the moment. For that reason, we have downloaded the page from the Google cache, and have attached it to this thesis as Appendix B.

socketcall (syscall 102)

socketcall is a multiplexed syscall; most or all of the socket operations are funneled through a single syscall number. The Linux man pages generally have documentation for the individual operations, but not how they are sent via socketcall. http://isoamerica.net/~dpn/socketcall1.pdf is an excellent resource for understanding socketcall.

[new]stat, [new]lstat, [new]fstat (syscalls 106,107,108)

The man pages define the `*stat` functions like this:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int stat(const char *path, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

Usually, the C library functions are simple wrappers of the syscalls. So, our original implementation of the syscall handler for these three syscalls copied `sizeof(struct stat)` bytes from the kernel to the user buffer. However, this caused segfaults every time that we called these functions through the C library.

It turns out that the internal kernel interface is different than what we see in the man pages. While `struct stat` is 88 bytes, it appears that the syscall only transfers 64 bytes. Thus, when we copied 88 bytes, we were overwriting 24 bytes in the C library function's stack. Apparently, the C library does some sort of translation, expanding the 64 bytes it gets from the kernel into the 88 bytes expected by users.

Our current implementation of these syscalls is hard-coded to only copy 64 bytes from the bounce buffer to the user buffer, and now it seems to work fine. However, we had to arrive at this number by experimentation; we did not find any documentation about it. (Thus, it might be different on other

[56] This author doesn't remember if it was reads or writes.

levels of Linux.)

MAP_FIXED

This was a useful feature that we overlooked early in the development of our shim. `mmap()` has a flag called `MAP_FIXED`. If you set this flag, and provide a recommended address for your map, Linux will atomically unmap any existing ranges and replace them with this one.

This was invaluable when we wanted to, in shim initialization, remap a page of code while we were running that very code. (Of course, we won't need this anymore, once the secure bootstrap method has been implemented.)

6.9 Summary

This chapter detailed our current implementation. It overviewed the basics of the implementation, including the Bochs base, and how the hypervisor hooks into Bochs to perform the necessary protections. It discussed the major structures and functions used in the current C code. It detailed how the shim is implemented. It closed with a TODO list, and a set of points of interest that we wanted to share with future implementers of similar systems.

Chapter 7: Results

Our current implementation implements effective runtime page protection. We are able to wrap a legacy binary application with our shim and protect its data from access by untrusted code. We have implemented wrappers for many system calls, and can run many of standard Linux tools in this mode. The protected processes are protected from both memory corruption and snooping, and yet are able to read input buffers from the kernel, perform processing on them, and output the results back to the public space. All of this is accomplished inside an entirely unmodified RedHat Linux 6.2 installation.

We cannot, at this time, report any meaningful performance numbers. Since it is a software emulator of the x86 architecture, Bochs is already quite slow, and therefore we cannot report any meaningful comparison between the time spent in the hypervisor (which runs at the full speed of the host machine) and time spent inside other code (which is emulated). We hope, in Future Work, to port our hypervisor to Xen or a similar VMM, at which point performance numbers will become meaningful.

7.1 Legacy Applications Run Correctly

Legacy applications still run correctly when they are wrapped with our shim. For example, we present the command `ls -al /`, running with and without the shim:

RUNNING NORMALLY

```
# ls -al /
total 89
drwxr-xr-x  16 root     root          1024 Sep 29 15:28 .
drwxr-xr-x  16 root     root          1024 Sep 29 15:28 ..
-rw-------   1 root     root            16 Sep 29 15:28 .bash_history
drwxr-xr-x   2 root     root          2048 Jun  1  2001 bin
drwxr-xr-x   3 root     root          1024 Jun  1  2001 boot
drwxr-xr-x   5 root     root         34816 Jan 21 12:07 dev
drwxr-xr-x  23 root     root          2048 Jan 21 12:07 etc
drwxr-xr-x   2 root     root          1024 Feb  6  1996 home
drwxr-xr-x   4 root     root          3072 Apr  8  2009 lib
drwxr-xr-x   2 root     root         12288 Jun  1  2001 lost+found
drwxr-xr-x   6 root     root          1024 Nov  6 04:33 mnt
dr-xr-xr-x  35 root     root             0 Jan 21 04:07 proc
drwxr-x---   9 root     root          1024 Jan 21 12:07 root
drwxr-xr-x   3 root     root          2048 Jun  1  2001 sbin
drwxrwxrwt   6 root     root         23552 Jan 21 12:12 tmp
drwxr-xr-x  19 root     root          1024 Apr  8  2009 usr
drwxr-xr-x  14 root     root          1024 Jun  1  2001 var
#
```

RUNNING UNDER SHIM

```
# LD_LIBRARY_PATH=$(pwd) LD_PRELOAD=shim2.so ls -al /
total 89
drwxr-xr-x  16 root     root          1024 Sep 29 15:28 .
drwxr-xr-x  16 root     root          1024 Sep 29 15:28 ..
```

```
-rw-------   1 root     root           16 Sep 29 15:28 .bash_history
drwxr-xr-x   2 root     root         2048 Jun  1  2001 bin
drwxr-xr-x   3 root     root         1024 Jun  1  2001 boot
drwxr-xr-x   5 root     root        34816 Jan 21 12:07 dev
drwxr-xr-x  23 root     root         2048 Jan 21 12:07 etc
drwxr-xr-x   2 root     root         1024 Feb  6  1996 home
drwxr-xr-x   4 root     root         3072 Apr  8  2009 lib
drwxr-xr-x   2 root     root        12288 Jun  1  2001 lost+found
drwxr-xr-x   6 root     root         1024 Nov  6 04:33 mnt
dr-xr-xr-x  36 root     root            0 Jan 21 04:07 proc
drwxr-x---   9 root     root         1024 Jan 21 12:07 root
drwxr-xr-x   3 root     root         2048 Jun  1  2001 sbin
drwxrwxrwt   6 root     root        23552 Jan 21 12:12 tmp
drwxr-xr-x  19 root     root         1024 Apr  8  2009 usr
drwxr-xr-x  14 root     root         1024 Jun  1  2001 var
#
```

We see that the command gives the exact same output, running with or without the shim.

7.2 Corruption Prevented

In order to test that our memory corruption code is working correctly, we must somehow simulate a corruption attack. We chose to do so by intentionally breaking the implementation of the `read()` syscall. We wrote code such that, on the third attempt to `read()`, we would “forget” to tell the hypervisor to discard the input buffer from out working set. Thus, when the kernel modifies those virtual pages and we later read them back, the hypervisor will detect corruption of the working set.

To test this, we ran the command `dd if=shim2.c bs=100` under our broken shim. This command will open the file `shim2.c` and then read 100 bytes at a time, writing it out to stdout. Below, you will see the result. Note that the `read() count=x` are debugging statements we added to the shim to help us understand where the `read()` syscalls are happening.

COMMAND OUTPUT

First, you see the command that we ran...

```
# LD_LIBRARY_PATH=$(pwd) LD_PRELOAD=shim2.so dd if=shim2.c bs=100
```

Next, we see the debug output for the first read()...

```
read() count=1
```

Then, the process writes that information out to stdout with the write() syscall...

```
#define _GNU_SOURCE

#include <unistd.h>

#include <sys/syscall.h>

#include <sys/mman.h>
```

```
#include <std
```

Then, the second read() and write()...

```
read() count=2
io.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <
```

Now, we see the third read(), which we intentionally fail...

```
read() count=3
Segmentation fault (core dumped)
#
```

The process segfaulted because the hypervisor detected corruption, destroyed the entity, and then sent a page fault to untrusted OS.

STRACE OUTPUT

We ran this test under strace, which gives us another perspective. Note that strace was **NOT** part of the protected entity; thus, when it looks at pages, it sees the same contents as does the kernel.

The output starts with a `write()` syscall for the debug `printf()` that we added.

```
write(1, "\nread() count=1\n", 16)      = 16
```

Next, you see that the `read()` syscall is actually performed. Note that the string that strace prints out as the 2nd argument is actually the contents of the buffer **AFTER** the read. We notice that it matches what is written out immediately afterward:

```
read(3, "#define _GNU_SOURCE\n#include <u"..., 100) = 100
write(1, "#define _GNU_SOURCE\n#include <u"..., 100) = 100
```

Another syscall, along with its debugging `printf()`:

```
write(1, "\nread() count=2\n", 16)      = 16
read(3, "io.h>\n#include <sys/types.h>\n#"..., 100) = 100
write(1, "io.h>\n#include <sys/types.h>\n#"..., 100) = 100
```

Notice what happens on the 3rd `read()`. We first get the debugging `printf()`. Then we get the `read()` syscall. Notice that strace, which is an unprotected process, reads the value of the buffer[57] **after** the syscall is performed, and sees the expected data – the data that the kernel read. However, the process hits SEGV right after the `read()` syscall returns:

```
write(1, "\nread() count=3\n", 16)      = 16

read(3, "sys/time.h>\n#include <sys/resou"..., 100) = 100

--- SIGSEGV (Segmentation fault) ---
```

HYPERVISOR LOGS

Finally, we look at our hypervisor logs. We see first hypercall 7, which tells the hypervisor that the next syscall should go through to the kernel, rather than getting redirected to the syscall handler...

```
07041078231i[CPU0 ] untrustedOS_interrupt_400_handler: RUSS: args: 7
(0x00000000,0x00000000,0x00000000,0x00000000) procID=0x00000000
```

Next, we see the read() syscall going to the kernel...

```
07041078264i[CPU0 ] syscall: 3 (read): 0x00000003, 0xbfeff000,
0x00000064

07041078264i[CPU0 ] untrustedOS_startInterrupt: Changing curProc to
0x00000000

07041078264i[CPU0 ] interrupt: Flushing the TLB
```

We notice that the kernel attempts to write a frame that currently has a logical page mapped. This is when the kernel actually “corrupts” that page. (Take note of the virtual address, 0xbfef_f000.)

```
07041082975i[CPU0 ] untrustedOS_writePage: Unmapping a logical page
because of an OS write (NOTE: readPage left it there, which means
that it must be public, or a hidden signature!)  vaddr=0xbfeff000
frame=0xb78de498
```

Next, the kernel jumps back to the address 0x08cb_d0b0, which is trusted code. We figure out that we are jumping back into the process, and change the `curProc` variable inside the hypervisor...

```
07041091572i[CPU0 ] FindInWorkingSet: proc=0x08cbd0b0
vaddr=0x400c8b9d add=0 frame=0xb78bfa88

07041091572i[CPU0 ] FindInWorkingSet: FOUND!!! Returning 0x08cc0dd0

07041091572i[CPU0 ] execPage_SetMask: CHANGING curProc TO 0x08cbd0b0
(old curProc=0x00000000)
```

...

Before long, the process attempts to read the virtual address 0xbfef_f000 (the page corrupted above), and we detect corruption...

```
07041091635i[CPU0 ] FindInWorkingSet: FOUND!!! Returning 0x08dd3998
```

[57] Technically, what strace is doing is using `ptrace()` to peek into the buffers of the protected process; the kernel will read the data out on strace's behalf, and copy it to strace's own buffers. So, what strace prints out is actually what the kernel perceives the contents of the page to be.

```
07041091635i[CPU0 ] untrustedOS_readPage: MISCOMPARE:
vaddr=0xbfeff000
```

SUMMARY

We see that if the kernel attempts to modify a virtual page of some protected process, and this process has not told the hypervisor to allow it, that the process will be killed as soon as it attempts to use that page.

7.3 Snooping Prevented

In order to confirm that snooping is prevented, we made a similar modification to our shim. This time, we chose that every 3rd `write()` syscall will "forget" to mark the buffer public. Thus, every 3rd `write()`, we expect that the kernel will see a signature page rather than the real data.

For easy comparison, we will wrap the shim around the same command as we used in the corruption example, `dd if=shim2.c bs=100`. We'll only print the first several lines of output, however. (Snooping does not cause the process to crash, so in this example, the command runs all the way to completion.)

COMMAND OUTPUT

```
# LD_LIBRARY_PATH=$(pwd) LD_PRELOAD=shim2.so dd if=shim2.c bs=100
```

First, we notice the debugging `printf()`s for the read, and then the write. Then we see the data printed to `stdout`:

```
read() count=1
write() count=1
#define _GNU_SOURCE
#include <unistd.h>
#include <sys/syscall.h>
#include <sys/mman.h>
#include <std
```

Then, it happens a second time, for the next block of data:

```
read() count=2
write() count=2
io.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```
#include <string.h>
#include <
```

However, when we get to the 3rd write, the shim “forgets” to mark the page public. Thus, the data that actually gets written to `stdout` is a signature. Notice the magic string “`untrusted OS signature page`”, and then random bytes:

```
read() count=3
write() count=3
untrusted OS signature page[begin random unprintable characters]
```

On the 4th iteration, the shim acts properly again, and again the kernel sees the correct data:

```
read() count=4
write() count=4
h>
#include <signal.h>
#include <sys/utsname.h>
#include <sys/vfs.h>
```

SUMMARY

We see that if the kernel attempts to access a virtual page of some protected process, and this process has not told the hypervisor that the page is public, then the kernel will see a signature instead of the real data.

7.4 Custom Test App

We coded a custom application to further demonstrate protection in action. It reads in a 4K buffer from stdin into a page-aligned buffer, ensures that it is a private page, and then writes the buffer out twice. In one case, it writes the buffer out to a file, using hypercall 7 to bypass the syscall handler. In the other case, it writes the buffer out with `fwrite()` to stdout, which uses the normal syscall intercept mechanism. This latter write is intercepted by the syscall handler, put on a public page, and then written to stdout. Since the bounce buffer a public page, the correct (private) values are written. Finally, it `mmap()`s the page it just wrote to the temporary file, and writes that buffer out with `fwrite()` as well. This third write also shows the correct, non-signature data, even though the file contents are the signature, rather than the private data.

What this illustrates is both implicit working set expansion and automatic restoration of signatures. As we will discover at the end of this experiment, the temporary file contains a signature, **NOT** the private data. However, when we `mmap()` a page of the temporary file to the protected process, it sees the private data (as evidenced by the last `fwrite()` output).

SOURCE CODE

```
#include <stdio.h>
#include <sys/mman.h>

int main()
{
  /* set up a page-aligned buffer to hold data */
  char  buf[2*4096];
  char *ptr = (char*)((((unsigned int)&buf[0])+4095) & ~4095);

  int len;

  char *mmap_ptr;

  /* set up a dump file */
  FILE *dump = fopen("dump", "w+");

  /* read up to one page from stdin */
  len = fread(ptr, 1,4096, stdin);
    ptr[len] = 0; /* null terminator */

  printf("len=%d\n", len);

  /* note that the buffer is already private data, because the
   * shim wrote to the buffer, copying the data from the bounce
   * buffer.
   */

  printf("BUFFER CONTENTS:\n");
  fwrite(ptr, len,1, stdout);

  /* write the buffer to the dump file.  However, send hypercall 7 to tell
   * the hypervisor to bypass the syscall handler.  Thus, the page will get
   * written as a private page (thus, a signature).
   */
  printf("WRITING RAW BUFFER TO FILE dump\n"); fflush(NULL);
  syscall(400,7);
  syscall(4, fileno(dump), ptr, 4096);
  /* next, write this same page to stdout, using the shim, which will use
   * a public bounce buffer page, so that the kernel will see the correct
   * data.
   */
   printf("WRITING BUFFER TO stdout USING SHIM, SO IT WILL SEE PRIVATE DATA\n");
fflush(NULL);
  fwrite(ptr, len,1, stdout);

  /* finally, we mmap() the page we just wrote to 'dump', and fwrite that.
   * What you will find written to stdout is actually the cleartext, even
   * though the page stored in 'dump' is a signature.
   */
  printf("MMAP()ing SIGNATURE PAGE\n"); fflush(NULL);
  mmap_ptr = mmap(NULL, 4096, PROT_READ, MAP_PRIVATE, fileno(dump), 0);

  printf("PRINTING RECENT mmap() TO stdout\n"); fflush(NULL);
  fwrite(mmap_ptr, len,1, stdout);

  return 0;
}
```

OUTPUT

Notice that the application reads data in from an input file. There are 46 characters in the file, spread across two lines. The application prints out what it saw.

```
len=46
BUFFER CONTENTS:
<this is the test file>
<there are two lines>
```

The application writes the raw buffer to the file 'dump', without using the shim. Note that under the covers, Bodyguard converts the private page to a signature before the OS is allowed to read it.

```
WRITING RAW BUFFER TO FILE dump
WRITING BUFFER TO stdout USING SHIM, SO IT WILL SEE PRIVATE DATA
```

The application writes the buffer to stdout again, again using the shim. Note that the shim copies data to a public bounce buffer before it actually performs the write. Observe that Bodyguard automatically converted the page back to the private data before the shim was allowed to perform the copy.

```
<this is the test file>
<there are two lines>
```

The application mmap()s the page which it just wrote to file 'dump'. As we see below, 'dump' contains the signature (**not** the private data), but when the application writes out that new buffer (again, using the shim), we see that it has the private data that we expect. What this means is that the hypervisor automatically signature (completely without any help from the application), and showed the private data to the application.

```
MMAP()ing SIGNATURE PAGE

PRINTING RECENT mmap() TO stdout

<this is the test file>

<there are two lines>
```

THE 'dump' FILE

We confirm that the contents of the dump are a signature:

```
untrusted OS signature pageK^_~X^?~D„]~QUEŠ$E~V^]@Iƒ^F=~OŒ
^PgqAI]rú"~QO(^U3~Eþ^

...<random unprintable characters>...
```

HYPERVISOR LOGS

We'll start looking at the logs at the point where the application opens up the 'dump' file. The syscall is intercepted by the syscall handler, sending it to the shim.

```
53405325691i[CPU0 ] untrustedOS_syscallAlert: syscallHandler=0x400172e4 allowNext=0

53405325691i[CPU0 ] syscall: 5 (open): 0x08048783, 0x00000242, 0x000001b6

53405325691i[CPU0 ] INT_Ib: The hypervisor told us to redirect this syscall to the
handler at address 0x400172e4
```

…

The shim sets up a bounce buffer for the file name, and calls hypercall 10 ("Make Range Public") to make the buffer at 0xbfdfd000 , length 0x00100000 into a public buffer.

```
53405329863i[CPU0   ]   untrustedOS_interrupt_400_handler:   RUSS:   args:   10
(0xbfdfd000,0x00100000,0x00000000,0x00000000) procID=0x00000000
```

…

The shim calls hypercall 7 ("Allow Next Syscall") and then performs the open syscall:

```
53405329923i[CPU0   ]   untrustedOS_interrupt_400_handler:   RUSS:   args:   7
(0x00000000,0x00000000,0x00000000,0x00000000) procID=0x00000000
```

…

```
53405329957i[CPU0 ] syscall: 5 (open): 0xbfdfd000, 0x00000242, 0x000001b6

53405329957i[CPU0 ] untrustedOS_startInterrupt: Changing curProc to 0x00000000

53405329957i[CPU0 ] interrupt: Flushing the TLB
```

…

Hypercall 14 is the hypercall which the syscall handler uses to unwrap the non-standard stack frame. The second argument to that hypercall is the return value, which Bochs needs to place in register EAX.

In this case, you see that the retval is 3. So, the file descriptor for the 'dump' file is 3.

```
53405346031i[CPU0 ] INT_Ib: Handling this without having to call
untrustedOS_interrupt_400_handler: args: 14 (0xbfffdc94, 0x00000003, …)
```

…

This is where the application uses syscall 7 to bypass the syscall handler, and then calls the write syscall to write the raw buffer out to the dump file. Notice that the first argument is the file descriptor (3); the second is the buffer address (note that it's on the stack and page aligned); the third is the length (0x1000, exactly one page).

```
53405480134i[CPU0 ] untrustedOS_interrupt_400_handler: RUSS: args: 7
(0x401081b4,0x080499a0,0xbfffe000,0x00000000) procID=0x00000000
```

…

```
53405480957i[CPU0 ] untrustedOS_syscallAlert: syscallHandler=0x400172e4 allowNext=1

53405480957i[CPU0 ] syscall: 4 (write): 0x00000003, 0xbfffe000, 0x00001000

53405480957i[CPU0 ] untrustedOS_startInterrupt: Changing curProc to 0x00000000

53405480957i[CPU0 ] interrupt: Flushing the TLB
```

…

When the kernel tries to read the private page, Bodyguard drops in a signature. Notice several things here. First, the kernel uses a different virtual address than the application, but the ppf (physical page address) matches the physical page address of the application's page 0xbfffe000. Notice the key which is generated for the signature; this is used to index a hash table of signatures, and will become important later. Finally, notice the state of the virtual page when we DUMP STATE after saving contents: you will see that the virtual page maps to a single logical page; the logical page is mapped into the same frame which the kernel was reading; the frame has its signature showing; the frame also has a SavedPage attached.

```
53405484187i[CPU0 ] untrustedOS_readPage: vaddr=0xc31c7000 ppf=0x031c7000

53405484187i[CPU0 ] SaveContents: RUSS: frame=0xb7970c78 logicalPage=0x08ed3e38
vaddr=0xbfffe000

53405484187i[CPU0 ] SaveContents: RUSS: sPage=0x08ed4ef8 key=0x456dae8f
```

…

```
53405484187i[CPU0 ] HidePrivateData: RUSS: frame=0xb7970c78 logicalPage=0x08ed3e38
vaddr=0xbfffe000

53405484187i[CPU0 ]

53405484187i[CPU0 ] DUMP STATE: (HidePrivateData 0xbfffe000)
------------------------------------------------------------------------------

53405484187i[CPU0 ]   curProc = 0x00000000
```

…

```
53405484187i[CPU0 ]       VPAGE: 0x08ed38c0

53405484187i[CPU0 ]         vaddr = 0xbfffe000

53405484187i[CPU0 ]         logicalPage = 0x08ed3e38

53405484187i[CPU0 ]           LPAGE: 0x08ed3e38

53405484187i[CPU0 ]             shared/priv = 0/1
```

```
53405484187i[CPU0 ]             frame = 0xb7970c78
53405484187i[CPU0 ]               ppf = 0x031c7000
53405484187i[CPU0 ]               buf = 0xb4a64000
53405484187i[CPU0 ]               logicalPage = 0x08ed3e38
53405484187i[CPU0 ]               sigShowing/sigScanNeeded = 1/0
53405484187i[CPU0 ]             savedPage = 0x08ed4ef8
53405484187i[CPU0 ]               alias = 0x00000000
53405484187i[CPU0 ]               owner = 0x08dbe020
53405484187i[CPU0 ]               useCount = 2
53405484187i[CPU0 ]               contents = 0xb35e8000
53405484187i[CPU0 ]               front = 0x00000000
53405484187i[CPU0 ]               back = 0x00001000
…
```

Now, we jump ahead to the point where we try to `mmap()` the page from the 'dump' file. The arguments to `mmap()` don't tell us much, since (as you can see from the syscall documentation in Appendix B) `mmap()` only has one argument; it points to a struct, which carries the various sub-arguments. In this case, the struct is on the stack, and the hypervisor logs don't give us details about its contents. However, we see it get redirected to the syscall handler, which puts the arguments in a bounce buffer, and then calls the kernel.

```
53406844121i[CPU0 ] untrustedOS_syscallAlert: syscallHandler=0x400172e4 allowNext=0
53406844121i[CPU0 ] syscall: 90 (old_mmap): 0xbfffdd14, 0x4001b000, 0xbfffe000
53406844121i[CPU0 ] INT_Ib: The hypervisor told us to redirect this syscall to the
handler at address 0x400172e4
…
53406847034i[CPU0 ] untrustedOS_interrupt_400_handler: RUSS: args: 10
(0xbfdfd000,0x00100000,0x00000000,0x00000000) procID=0x00000000
…
53406847094i[CPU0 ] untrustedOS_interrupt_400_handler: RUSS: args: 7
(0x00000000,0x00000000,0x00000000,0x00000000) procID=0x00000000
…
53406847123i[CPU0 ] untrustedOS_syscallAlert: syscallHandler=0x400172e4 allowNext=1
53406847123i[CPU0 ] syscall: 90 (old_mmap): 0xbfdfd000, 0x00000000, 0x00000000
53406847123i[CPU0 ] untrustedOS_startInterrupt: Changing curProc to 0x00000000
53406847123i[CPU0 ] interrupt: Flushing the TLB
…
```

When we perform hypercall 14 to return from the syscall, we see that the return value from `mmap()` was 0x4001c000.

```
53406864796i[CPU0 ] INT_Ib: Handling this without having to call
untrustedOS_interrupt_400_handler: args: 14 (0xbfffdcdc, 0x4001c000, …)
```

…

Now, the first time that we touch page 0x4001c000, FindInWorkingSet implicitly adds the page to the working set. It compares the contents to our list of signatures, and find a match to the page we saved before. (The key for the signature, read from the frame, matches the key for the signature we generated above, which allows us to find the old signature in the hash table.) Then, we DUMP STATE again, and we can see that the two virtual pages (0x4001c000 and 0xbfffe000) both point to the same SavedPage.

```
53406891017i[CPU0 ] untrustedOS_readPage: vaddr=0x4001c000 ppf=0x01d12000
53406891017i[CPU0 ] FindInWorkingSet: proc=0x08dbe088 vaddr=0x4001c000 add=1
frame=0xb795c128
53406891017i[CPU0 ] FindInWorkingSet: ADDING virt addr 0x4001c000 to the working
set!
53406891017i[CPU0 ] ScanForSignatures: RUSS: ppf=0x01d12000 key=0x456dae8f
53406891017i[CPU0 ] ScanForSignatures: RUSS: memcmp sPage=0x08ed4ef8 key=0x456dae8f
53406891017i[CPU0 ] ScanForSignatures: RUSS: FOUND!!! sPage=0x08ed4ef8
53406891017i[CPU0 ] FindInWorkingSet: ---> Found that the new page matches a saved
signature
53406891017i[CPU0 ] ShowPrivateData: RUSS: frame=0xb795c128 logicalPage=0x08ee47d8
vaddr=0x4001c000
53406891017i[CPU0 ]
53406891017i[CPU0 ] DUMP STATE: (ShowPrivateData 0x4001c000)
------------------------------------------------------------------------------
```

…

```
53406891017i[CPU0 ]       VPAGE: 0x08ee4260
53406891017i[CPU0 ]         vaddr = 0x4001c000
53406891017i[CPU0 ]         logicalPage = 0x08ee47d8
53406891017i[CPU0 ]           LPAGE: 0x08ee47d8
53406891017i[CPU0 ]             shared/priv = 0/1
53406891017i[CPU0 ]             frame = 0xb795c128
53406891017i[CPU0 ]               ppf = 0x01d12000
53406891017i[CPU0 ]               buf = 0xb5daf000
53406891017i[CPU0 ]               logicalPage = 0x08ee47d8
53406891017i[CPU0 ]               sigShowing/sigScanNeeded = 0/0
53406891017i[CPU0 ]             savedPage = 0x08ed4ef8
53406891017i[CPU0 ]               alias = 0x00000000
53406891017i[CPU0 ]               owner = 0x08dbe020
53406891017i[CPU0 ]               useCount = 3
53406891017i[CPU0 ]               contents = 0xb35e8000
53406891017i[CPU0 ]               front = 0x00000000
53406891017i[CPU0 ]               back = 0x00001000
```

…

```
53406891017i[CPU0 ]       VPAGE: 0x08ed38c0
53406891017i[CPU0 ]         vaddr = 0xbfffe000
53406891017i[CPU0 ]         logicalPage = 0x08ed3e38
53406891017i[CPU0 ]           LPAGE: 0x08ed3e38
53406891017i[CPU0 ]             shared/priv = 0/1
53406891017i[CPU0 ]             frame = 0xb7970c78
53406891017i[CPU0 ]               ppf = 0x031c7000
53406891017i[CPU0 ]               buf = 0xb4a64000
53406891017i[CPU0 ]               logicalPage = 0x08ed3e38
53406891017i[CPU0 ]               sigShowing/sigScanNeeded = 0/0
53406891017i[CPU0 ]             savedPage = 0x08ed4ef8
53406891017i[CPU0 ]               alias = 0x00000000
53406891017i[CPU0 ]               owner = 0x08dbe020
53406891017i[CPU0 ]               useCount = 3
53406891017i[CPU0 ]               contents = 0xb35e8000
53406891017i[CPU0 ]               front = 0x00000000
53406891017i[CPU0 ]               back = 0x00001000
```

…

SUMMARY

We have seen that, if we write a private page to disk without using a bounce buffer, the untrusted OS will actually write the signature instead of the private data. Moreover, if we later `mmap()` that page into a new virtual page, the hypervisor automatically detects that the page is a duplicate of an old signature and shows the private data to the application.

7.5 Chapter Summary

We have run several testcases which demonstrate that the page protection mechanisms are functioning properly. The signature mechanism prevents snooping; the page verification mechanism prevents corruption; the signature recognition mechanism automatically recognizes signatures in newly-mapped pages, and converts them to private data.

Chapter 8: Conclusion

8.1 Future Work

Chapter 4 (Bodyguard Design) detailed our current design for this system. Although it is not fully implemented yet (see Section 6.7: TODO List), we believe that the design offers efficient and complete protection for almost all applications.

This section goes further, and details changes to the design that we may pursue in the future.

Xen

We hope to port our hypervisor to Xen or a similar VMM. While Bochs was easy to modify for our initial implementation, its performance is far from adequate. By porting the hypervisor to a more practical VMM, we hope to be able to eventually deploy the hypervisor within ordinary environments, and make it available as a standard feature on some machines.

To that end, the hypervisor was written with porting in mind; we have a fairly clean interface between the hypervisor and the rest of the code.

Generalization

We expect that this design will work well, with minor modifications, on Windows and other operating systems. For instance, the hypercall mechanism may need to be changed. Likewise, the list of registers that are protected across interrupts and syscalls may vary from OS to OS.

We expect to add configuration parameters in the client/hypervisor communication that would allow a single hypervisor to be dynamically configured to support the conventions of any guest OS.

Better Automatic Protection

As noted in Section 4.12.1: Limitations, there are some fundamental limitations to how much a shim can do automatically. For instance, the shim cannot know, with absolute certainty, whether file output contains private data, which should be protected, or public data, which should be written for untrusted entities to read.

Future investigators may develop better heuristics for choosing default values and/or may be able to find some way overcome these obstacles altogether.

Automatic Private Files

The shim needs to automate the process of writing, and then later verifying, the contents of private files. While the hypervisor automatically protects the current working set, it cannot automatically verify the contents of newly-`mmap()`ed pages. Thus, when a process `mmap()`s or `read()`s a file containing private data, it needs to verify that the data read is actually the data that ought to exist in that file.

We imagine that this could be accomplished with a system of checksums. Our proposed design assumes that the first page of a private file would contain a header with a secret magic value (so that the untrusted OS would not be able to falsify such a page). It would also include a field describing the real length of the file, and then hold the checksums of the next N pages. If the size of the underlying file (not counting checksum overhead) exceeded N-1 pages, then the last page of the N pages would contain another checksum table, for the next many pages.[58] Thus, if the header page can be verified, and all of the chains of checksums are valid, then the entire file contents are valid.

We did not add this to our current design, however, because there are a number of unresolved questions. First, how do we prevent stale files from being presented? We could add a sequence number to the file header in order to keep track of the latest version of each file, but then how do we map file descriptors to the expected sequence numbers? Also, can this system be adapted to work with executable files, where the header needs to be untrusted-OS-readable?

Hash-Based Protection

When a process needs to verify the contents of a recent `mmap()`, the current design requires that it read the page immediately. However, this is inelegant and also may be undesirable for performance (for instance, we may not want to read the page up from disk until it is needed). A simple improvement would be a "register expected checksum" hypercall. This would give the expected checksum of a recently-mapped page (one which was not yet in the working set). The hypervisor would retain this checksum until the page is actually accessed at some point in the future, and would use the checksum to verify the initial contents. (After the initial verification, normal hypervisor mechanisms, with saved pages and signatures, would be used.)

Hybrid Register Protection Scheme

Both Overshadow and Bodyguard expect that the hypervisor needs to play a role in register verification and restoration when untrusted code returns into a protected process. They implement this with a trampoline and CTCs in virtual memory, where as we implement this with implicit jump detection and registers stored in hypervisor memory.

A more attractive option would be to move all of this verification into the shim. In this proposed design, we use CTCs like in Overshadow, but the hypervisor does not verify calls into the protected process. Instead, when the hypervisor detects a jump into the protected process, it forces the code to a handler inside that process (analogous to the syscall handler). This handler would verify that the register values, as set up by the kernel, match what is expected. This handler would also be responsible for restoring thread state. Finally, the handler would be responsible for informing the hypervisor where to store the thread state whenever this process calls out to untrusted code again.

We like this idea because it simplifies the hypervisor; however, we know that this handler will be quite difficult to write. For instance, it must be able to survive page faults inside the handler.

Signal Handling

The current design does not account for signals. In order to support all programs, we must add this support. However, if we implement the hybrid register protection scheme (see above), then it may be possible to implement this entirely in the shim, with no hypervisor support at all.

Better Process Identification

One advantage of Overshadow's trampoline-based system is that each process (and each thread within the process) has a clear value, saved in virtual memory, which uniquely identifies which process and thread this is. Of course, an attacker could corrupt this value, but this would simply resolve to a DOS

[58] Perhaps the 2nd table should be a two-layer table, and so on, for faster indexing? The system, as described above, takes linear time to look up the expected hash for a page; with layered tables, we could make that $O(\log n)$. I suggest that we have each progressive table be deeper than the last so that the front end of a very long file would have the same format (one-layer table) as the contents of a short file. That makes appending and cropping much easier.

attack when the trampoline is unable to re-activate a thread.

The fundamental advantage is that a well-meaning OS is not going to change this value (since it resides in virtual space). Our design, on the other hand, uses the page table pointer as a process ID; while we haven't seen it change, it certainly might, under unknown circumstances.

Future investigators should look for a more reliable process ID. Ideally, we want a process ID that is still compatible with our design that implicitly detects when we jump into a protected process. However, it is also possible for us to adopt an Overshadow-style trampoline.

Discard Old Signatures

Our current design requires that we store all signatures, for all saved pages, for the entire life of the entity. This is because the hypervisor doesn't have any way to know which pages might have been written to disk, and might later be restored.

Future investigators could pursue methods for determining when old signatures can be discarded.

Shared Memory for IPC

When two processes in the same entity communicate using IPC, there is the potential for both corruption and snooping. One option would be to encrypt all data, using checksums and sequence numbers to validate the decrypted data.

We prefer, however, the idea of using shared memory for IPC. In this idea, all of the communicating processes would map a set of shared private pages, which would hold buffers to exchange. The IPC mechanisms in the untrusted OS would then be used only for signaling when buffers are ready. The untrusted OS's only possible attack, then, would be to fail to deliver the wakeup signals.

8.2 Close

Even the most carefully coded and well tested application is vulnerable to attacks through the operating system. An attacker that has control of the operating system has the ability to snoop on private data, corrupt results, or simply deny resources to the application.

In this thesis we have presented a system that makes it impossible for an attacker, even with complete control of the kernel, to either snoop on private data or corrupt the results. Moreover, we have discussed how, given these protections, the client may reliably monitor the status of the application and detect DOS attacks. All of this is accomplished with a very simple hypervisor, provided that the application is modified to take advantage of the hypervisor features. Finally, we presented a shim that can automatically modify a legacy application to take advantage of the hypervisor features.

We believe that this system is a practical method for ensuring application correctness and privacy against arbitrary attacks. It forms an important extra layer of security for critical applications.

Bibliography

[1] T. Ball, S. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. POPL '02.

[2] D. Engler, M. Musuvathi. Static analysis versus software model checking for bug finding. Stanford University.

[3] T. Ball, R. Majumdar, T. Millstein, S. Rajamani. Automatic Predicate Abstraction of C Programs. PLDI 2001.

[4] D. Hovemeyer, J. Spacco, W. Pugh. Evaluating and Tuning a Static Analysis to Find Null Pointer Bugs. PASTE '05.

[5] S. Hallem, B. Chelf, Y. Xie, D. Engler. A System and Language for Building System Specific Static Analyses. PLDI '02.

[6] D. Kienzle, M. Elder. Recent Worms: A Survey and Trends. WORM '03.

[7] C. Landwehr, A. Bull, J. McDermott, W. Choi. A Taxonomy of Computer Program Security Flaws, with Examples. ACM Computing Surveys 26, 3 (Sept 1994).

[8] A. Burdonov, A. Kosachev, P. Iakovenko. Virtualization-based separation of privilege: working with sensitive data in untrusted environment. VTDS 2009.

[9] S. Kamara, S. Fahmy, E. Schultz, F. Kerschbaum, M. Frantzen. Analysis of Vulnerabilities in Internet Firewalls. Center for Education and Research in Information Assurance and Security (CERIAS) Purdue University.

[10] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, D. Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. SOSP '03.

[11] P. Loscocco, S. Smalley, P. Muckelbauer, R. Taylor, S. Turner, J. Farrell. The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments. NSA.

[12] V. Haldar, D. Chandra, M. Franz. Semantic Remote Attestation —A Virtual Machine directed approach to Trusted Computing. Department of Computer Science, University of California.

[13] N. Zeldovich, H. Kannan, M. Dalton, C. Kozyrakis. Hardware Enforcement of Application Security Policies Using Tagged Memory. 8th USENIX Symposium on Operating Systems Design and Implementation.

[14] R. Lee, P. Kwan, J. McGregor, J. Dwoskin, Z. Wang. Architecture for Protecting Critical Secrets in Microprocessors. ISCA '05.

[15] J. Dwoskin, R. Lee. Hardware-rooted Trust for Secure Key Management and Transient Trust. CCS '07.

[16] Y. Chen, R. Lee. Hardware-Assisted Application-Level Access Control. Information Security Conference, Sep 7-9, 2009.

[17] J. Dyer, M. Lindermann, R. Perez, R. Sailer, L. van Doorn, S. Smith, S. Weingart. Building the IBM 4758 Secure Coprocessor. "Computer" Magazine, October 2001, pp. 57-66.

[18] A. Seshadri, M. Luk, N. Qu, A. Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. SOSP '07.

[19] J. Franklin, A. Seshadri, N. Qu, S. Chaki, A. Datta. Attacking, Repairing, and Verifying SecVisor: A Retrospective on the Security of a Hypervisor. Carnegie Mellon University.

[20] E. Bugnion, S. Devine, K. Govil, M. Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. SOSP-16.

[21] J. Li, M. Krohn, D. Mazi`eres, D. Sasha. Secure Untrusted Data Repository (SUNDR). OSDI '04: 6th Symposium on Operating Systems Design and Implementation.

[22] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hagegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono. BitVisor: A Thin Hypervisor for Enforcing I/O Device Security. VEE '09.

[23] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho,R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. SOSP '03.

[24] D. Murray, S. Hand. Privilege separation made easy: Trusting small libraries not big processes. EuroSec '08.

[25] X. Chen, T. Garfinkel, E. Lewis, P. Subrahmanyam, C. Waldspurger, D. Boneh, J. Dwoskin, D. Ports. Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. ASPLOS '08.

[26] http://www.vmware.com/

[27] R. Riley, X. Jiang, D. Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. RAID 2008.

[28] http://bochs.sourceforge.net, http://en.wikipedia.org/wiki/Bochs

Appendix A: Glossary

Attacker – a malicious agent, who may have complete knowledge of the untrusted OS, the trusted application, and any protection mechanisms implemented by the hypervisor, and that may have compromised the untrusted OS long before the client attempts to run the trusted application.

Client – an external computer (either a trusted guest OS running under the same hypervisor as the untrusted OS, or a remote computer) that is assumed to be not under the control of an attacker. When the client wants to run a new instance of a trusted application, it contacts the hypervisor through a secure channel and asks it to create a new entity; it then uses that channel to communicate the expected initial state of that entity to the hypervisor. Later, it contacts the untrusted OS and runs the application; the hypervisor ensures that the initial state of the application, as loaded by the OS, matches what the client told it to expect.

Corruption – an attack class where an attacker attempts to change the state of the process without causing an obvious DOS event, such as a segfault. This paper attempts to detect all such attacks and convert them to program crashes.

DOS (Denial of Service) – an attack class where an attacker denies a critical resource to the protected application. This could involve rejecting allocation requests, failing to provide promised resources, or taking away promised resources. This paper assumes that the client uses timeouts or heartbeating to ensure that the protected process is at least getting time on the processor (and network resources to communicate with the outside world), and expects that other forms of DOS can be detected, and reported, by the protected process itself.

Entity – an instance of a protected application, comprising one or more processes running under a certain untrusted OS. Other entities, even other instances of the same trusted application, may never view the private data of any process in this entity, nor corrupt the state of any of them. Likewise, code running outside of any known entity may never view the private data of any process that is part of any entity.

Frame – see Physical Page.

Hypercall – a call from a protected process into the hypervisor, bypassing any operating system mechanisms. Depending on the implementation, the operating system may or may not be aware of the hypercall and its arguments; in all implementations, the operating system is unable to prevent, alter, or falsify a hypercall.

Hypervisor – a hardware or software entity that has the primary responsibility of implementing the protections described in this paper. It keeps track of the list of entities, their processes, and their working sets, and prevents both corruption and snooping by any potential attacker. Typically, this is a VMM (Virtual Machine Monitor), but one might imagine a hardware implementation as well. In our work, we modify Bochs (a software emulator of the x86 architecture) to provide an emulated hardware platform that implements the hypervisor functionality.

Kernel/Kernel Code – A loose term for untrusted code running inside a protected process. (See also Untrusted Code.) When the application makes a system call (or an interrupt or exception occurs), we expect to be running kernel code inside our protected process. Taken more generally, though, we sometimes use the term "kernel code" to refer to **any** untrusted code that the kernel happens to map into our protected process. (The hypervisor has no way to distinguish between true "kernel" code that is appropriately mapped into our process and some other "bad" code that was maliciously mapped in.)

Logical Page – the logical contents of some particular page. The hypervisor keeps track of the expected contents of each logical page, as well as which physical page it may be mapped into. Typically, each logical page stores the contents of a single virtual page, but if the logical page represents a shared page, many virtual pages may map to it.

Physical Page/Physical Frame – a physical page (or frame) is a guest physical page. Sometimes, a physical page may be associated with one or more logical pages, meaning that there exist one or more virtual pages (within protected processes) that map to this physical page. In that context, the physical page may contain the private data (cleartext), or it may contain a signature (the obscured, public version of the page). In this thesis, the relationship between a frame and its various logical page(s) is updated lazily.

Private Page – any page that is part of a process's working set but the process has not yet declared public.

Process – a process, as it is known to the untrusted OS. If the process is a protected process, meaning that it is part of an entity, then the hypervisor also knows about this process, and keeps track of its working set.

Protected Application – see Trusted Application.

Protected Process – a process that is part of an entity. The protected process has the capability to run both private code pages and public code pages (that is, code pages authored by the untrusted OS or other applications). When we say that "the protected process does X," we are explicitly stating that it was running private code pages belonging to the entity. When the process happens to be running other code pages (such as kernel pages mapped into the address space), then we say that the "kernel is running" or "untrusted code is running," not that the protected process is doing so. (In this context, "kernel" is a loose term referring to any code not authored by that protected process, whether or not it happens to actually have supervisor privileges.) Untrusted code running inside a protected process gets no special privileges; it is subject to the same limitations as kernel code in any other untrusted process. In particular, when it attempts to access private pages that happen to be mapped into the process, it will not see the private data; it will see signatures instead. Likewise, writes by such code to private pages will not counted as valid changes to the process's working set. If such changes are later exposed to private code, this will be detected as a form of corruption, and will result in a program crash.

Public Page – a page in the working set of some protected process that the process has declared is safe to expose to untrusted code.

Signature – a page-sized block of data, comprising a magic string followed by random data, which uniquely identifies the private contents of some private page, at some point in time. When any code outside a given entity attempts to access any private page of that entity, the hypervisor steps in and replaces the page with a signature before allowing access. Later, when the entity attempts to access its own private data, the hypervisor will restore the private data. The hypervisor stores a list of all signatures created over the lifetime of a given entity, and allows new virtual pages to be initialized with an old signature. In such a case, the hypervisor automatically recognizes the signature, treats the page as a COW copy of the older private page, and restores the private data upon its first use by the entity.

Snooping – an attack class where the attacker attempts to read the private data of a protected application. This paper prevents all such attacks by replacing the page with a signature before the read may be attempted. Also includes attacks where the attacker may modify one or more bytes in the page and then attempt to read the contents; again, the hypervisor replaces the page with a signature before the write is allowed, meaning that the attacker will see the signature page, plus his modifications, as the contents.

Trusted Application – an application that has been debugged and further has been modified to fit the requirements of this thesis. It is presumed that this application, if its state is not corrupted, will always produce a correct result given arbitrary input. (In this definition, input includes OS action, including providing, failing to provide, or even corrupting, requested resources. "Correct result," in the face of operating system attack or failure, would constitute a report of the error to the client, or a heartbeat failure detected by the client. See also Client.) An instance of the application, running on a certain hypervisor, is known as an entity, and includes a plurality of interacting processes.

Untrusted Code – Code that is attempting to access the private data of some entity, but does not belong to the entity that owns it. This could be code that is not part of any entity, or that is part of a different entity. The code will see the signature for the page, rather than the private data.

Untrusted OS – a commodity OS, entirely unmodified, that we presume has some unknown number of vulnerabilities, which make it possible for the attacker to gain control[59] of the OS. We desire to run an instance of a trusted application under this OS, but we assume that unknown attackers may have complete control over the untrusted OS before we can even start this new application. See also Attacker.

Virtual Page – a single page in the context of some process. Specifically, a page in the working set of a protected process. There is a many-to-one relationship between virtual pages and logical pages. By default, each virtual page maps to a unique logical page. However, when the virtual pages represent shared pages, there will be a single logical page that represents the shared page, and many virtual pages, from many processes, will map to it. In rare circumstances, there may be a logical page that has no virtual pages that map to it; this represents a shared page that currently is not mapped into any virtual page, but whose contents the hypervisor is tracking so that it may validate the contents if the page is reused in the future.

Working Set – a list of virtual pages known by the hypervisor to be in the virtual address space of a protected process. This list is lazily updated, meaning that newly mmap()ed pages are not known by the hypervisor until the first time that the protected process accesses them. Note that this list also does not account for which pages are currently in main memory; any portion of the working set might be swapped out at any time.

[59] Either kernel control, or root access, which amounts to the same thing

Appendix B: Linux Syscall Table

This file was originally located at http://bluemaster.iu.hio.no/edu/ca/lin-asm/syscalls.html. At time of writing, it now appears to be offline, so we downloaded this file from the Google cache and have included it for future reference.

BEGIN INCLUDED FILE

Linux System Call Table

The following table lists the system calls for the Linux 2.2 kernel. It could also be thought of as an API for the interface between user space and kernel space. My motivation for making this table was to make programming in assembly language easier when using only system calls and not the C library (for more information on this topic, go to http://www.linuxassembly.org). On the left are the numbers of the system calls. This number will be put in register %eax. On the right of the table are the types of values to be put into the remaining registers before calling the software interrupt 'int 0x80'. After each syscall, an integer is returned in %eax.

For convenience, the kernel source file where each system call is located is linked to in the column labeled "Source". In order to use the hyperlinks, you must first copy this page to your own machine because the links take you directly to the source code on your system. You must have the kernel source installed (or linked from) under '/usr/src/linux' for this to work.

%eax	Name	Source	%ebx	%ecx	%edx	%esx	%edi
1	sys_exit	kernel/exit.c	int	-	-	-	-
2	sys_fork	arch/i386/kernel/process.c	struct pt_regs	-	-	-	-
3	sys_read	fs/read_write.c	unsigned int	char *	size_t	-	-
4	sys_write	fs/read_write.c	unsigned int	const char *	size_t	-	-
5	sys_open	fs/open.c	const char *	int	int	-	-
6	sys_close	fs/open.c	unsigned int	-	-	-	-
7	sys_waitpid	kernel/exit.c	pid_t	unsigned int *	int	-	-
8	sys_creat	fs/open.c	const char *	int	-	-	-
9	sys_link	fs/namei.c	const char *	const char *	-	-	-
10	sys_unlink	fs/namei.c	const char	-	-	-	-

			*				
11	sys_execve	arch/i386/kernel/process.c	struct pt_regs	-	-	-	-
12	sys_chdir	fs/open.c	const char *	-	-	-	-
13	sys_time	kernel/time.c	int *	-	-	-	-
14	sys_mknod	fs/namei.c	const char *	int	dev_t	-	-
15	sys_chmod	fs/open.c	const char *	mode_t	-	-	-
16	sys_lchown	fs/open.c	const char *	uid_t	gid_t	-	-
18	sys_stat	fs/stat.c	char *	struct __old_kernel_stat *	-	-	-
19	sys_lseek	fs/read_write.c	unsigned int	off_t	unsigned int	-	-
20	sys_getpid	kernel/sched.c	-	-	-	-	-
21	sys_mount	fs/super.c	char *	char *	char *	-	-
22	sys_oldumount	fs/super.c	char *	-	-	-	-
23	sys_setuid	kernel/sys.c	uid_t	-	-	-	-
24	sys_getuid	kernel/sched.c	-	-	-	-	-
25	sys_stime	kernel/time.c	int *	-	-	-	-
26	sys_ptrace	arch/i386/kernel/ptrace.c	long	long	long	long	-
27	sys_alarm	kernel/sched.c	unsigned int	-	-	-	-
28	sys_fstat	fs/stat.c	unsigned int	struct __old_kernel_stat *	-	-	-
29	sys_pause	arch/i386/kernel/sys_i386.c	-	-	-	-	-
30	sys_utime	fs/open.c	char *	struct utimbuf *	-	-	-
33	sys_access	fs/open.c	const char *	int	-	-	-
34	sys_nice	kernel/sched.c	int	-	-	-	-
36	sys_sync	fs/buffer.c	-	-	-	-	-

37	sys_kill	kernel/signal.c	int	int	-	-	-
38	sys_rename	fs/namei.c	const char *	const char *	-	-	-
39	sys_mkdir	fs/namei.c	const char *	int	-	-	-
40	sys_rmdir	fs/namei.c	const char *	-	-	-	-
41	sys_dup	fs/fcntl.c	unsigned int	-	-	-	-
42	sys_pipe	arch/i386/kernel/sys_i386.c	unsigned long *	-	-	-	-
43	sys_times	kernel/sys.c	struct tms *	-	-	-	-
45	sys_brk	mm/mmap.c	unsigned long	-	-	-	-
46	sys_setgid	kernel/sys.c	gid_t	-	-	-	-
47	sys_getgid	kernel/sched.c	-	-	-	-	-
48	sys_signal	kernel/signal.c	int	__sighandler_t	-	-	-
49	sys_geteuid	kernel/sched.c	-	-	-	-	-
50	sys_getegid	kernel/sched.c	-	-	-	-	-
51	sys_acct	kernel/acct.c	const char *	-	-	-	-
52	sys_umount	fs/super.c	char *	int	-	-	-
54	sys_ioctl	fs/ioctl.c	unsigned int	unsigned int	unsigned long	-	-
55	sys_fcntl	fs/fcntl.c	unsigned int	unsigned int	unsigned long	-	-
57	sys_setpgid	kernel/sys.c	pid_t	pid_t	-	-	-
59	sys_olduname	arch/i386/kernel/sys_i386.c	struct oldold_utsname *	-	-	-	-
60	sys_umask	kernel/sys.c	int	-	-	-	-
61	sys_chroot	fs/open.c	const char *	-	-	-	-
62	sys_ustat	fs/super.c	dev_t	struct ustat *	-	-	-
63	sys_dup2	fs/fcntl.c	unsigned int	unsigned int	-	-	-

64	sys_getppid	kernel/sched.c	-	-	-	-	-
65	sys_getpgrp	kernel/sys.c	-	-	-	-	-
66	sys_setsid	kernel/sys.c	-	-	-	-	-
67	sys_sigaction	arch/i386/kernel/signal.c	int	const struct old_sigaction *	struct old_sigaction *	-	-
68	sys_sgetmask	kernel/signal.c	-	-	-	-	-
69	sys_ssetmask	kernel/signal.c	int	-	-	-	-
70	sys_setreuid	kernel/sys.c	uid_t	uid_t	-	-	-
71	sys_setregid	kernel/sys.c	gid_t	gid_t	-	-	-
72	sys_sigsuspend	arch/i386/kernel/signal.c	int	int	old_sigset_t	-	-
73	sys_sigpending	kernel/signal.c	old_sigset_t *	-	-	-	-
74	sys_sethostname	kernel/sys.c	char *	int	-	-	-
75	sys_setrlimit	kernel/sys.c	unsigned int	struct rlimit *	-	-	-
76	sys_getrlimit	kernel/sys.c	unsigned int	struct rlimit *	-	-	-
77	sys_getrusage	kernel/sys.c	int	struct rusage *	-	-	-
78	sys_gettimeofday	kernel/time.c	struct timeval *	struct timezone *	-	-	-
79	sys_settimeofday	kernel/time.c	struct timeval *	struct timezone *	-	-	-
80	sys_getgroups	kernel/sys.c	int	gid_t *	-	-	-
81	sys_setgroups	kernel/sys.c	int	gid_t *	-	-	-
82	old_select	arch/i386/kernel/sys_i386.c	struct sel_arg_struct *	-	-	-	-
83	sys_symlink	fs/namei.c	const char *	const char *	-	-	-
84	sys_lstat	fs/stat.c	char *	struct __old_kernel_stat *	-	-	-
85	sys_readlink	fs/stat.c	const char *	char *	int	-	-
86	sys_uselib	fs/exec.c	const char	-	-	-	-

			*				
87	sys_swapon	mm/swapfile.c	const char *	int	-	-	-
88	sys_reboot	kernel/sys.c	int	int	int	void *	-
89	old_readdir	fs/readdir.c	unsigned int	void *	unsigned int	-	-
90	old_mmap	arch/i386/kernel/sys_i386.c	struct mmap_arg_struct *	-	-	-	-
91	sys_munmap	mm/mmap.c	unsigned long	size_t	-	-	-
92	sys_truncate	fs/open.c	const char *	unsigned long	-	-	-
93	sys_ftruncate	fs/open.c	unsigned int	unsigned long	-	-	-
94	sys_fchmod	fs/open.c	unsigned int	mode_t	-	-	-
95	sys_fchown	fs/open.c	unsigned int	uid_t	gid_t	-	-
96	sys_getpriority	kernel/sys.c	int	int	-	-	-
97	sys_setpriority	kernel/sys.c	int	int	int	-	-
99	sys_statfs	fs/open.c	const char *	struct statfs *	-	-	-
100	sys_fstatfs	fs/open.c	unsigned int	struct statfs *	-	-	-
101	sys_ioperm	arch/i386/kernel/ioport.c	unsigned long	unsigned long	int	-	-
102	sys_socketcall	net/socket.c	int	unsigned long *	-	-	-
103	sys_syslog	kernel/printk.c	int	char *	int	-	-
104	sys_setitimer	kernel/itimer.c	int	struct itimerval *	struct itimerval *	-	-
105	sys_getitimer	kernel/itimer.c	int	struct itimerval *	-	-	-
106	sys_newstat	fs/stat.c	char *	struct stat *	-	-	-
107	sys_newlstat	fs/stat.c	char *	struct stat *	-	-	-
108	sys_newfstat	fs/stat.c	unsigned int	struct stat *	-	-	-

109	sys_uname	arch/i386/kernel/sys_i386.c	struct old_utsname *	-	-	-	-	
110	sys_iopl	arch/i386/kernel/ioport.c	unsigned long	-	-	-	-	
111	sys_vhangup	fs/open.c	-	-	-	-	-	
112	sys_idle	arch/i386/kernel/process.c	-	-	-	-	-	
113	sys_vm86old	arch/i386/kernel/vm86.c	unsigned long	struct vm86plus_struct *	-	-	-	
114	sys_wait4	kernel/exit.c	pid_t	unsigned long *	int options	struct rusage *	-	
115	sys_swapoff	mm/swapfile.c	const char *	-	-	-	-	
116	sys_sysinfo	kernel/info.c	struct sysinfo *	-	-	-	-	
117	sys_ipc (*Note)	arch/i386/kernel/sys_i386.c	uint	int	int	int	void *	
118	sys_fsync	fs/buffer.c	unsigned int	-	-	-	-	
119	sys_sigreturn	arch/i386/kernel/signal.c	unsigned long	-	-	-	-	
120	sys_clone	arch/i386/kernel/process.c	struct pt_regs	-	-	-	-	
121	sys_setdomainname	kernel/sys.c	char *	int	-	-	-	
122	sys_newuname	kernel/sys.c	struct new_utsname *	-	-	-	-	
123	sys_modify_ldt	arch/i386/kernel/ldt.c	int	void *	unsigned long	-	-	
124	sys_adjtimex	kernel/time.c	struct timex *	-	-	-	-	
125	sys_mprotect	mm/mprotect.c	unsigned long	size_t	unsigned long	-	-	
126	sys_sigprocmask	kernel/signal.c	int	old_sigset_t *	old_sigset_t *	-	-	
127	sys_create_modu	kernel/module	const char	size_t	-	-	-	

	le	.c	*				
128	sys_init_module	kernel/module.c	const char *	struct module *	-	-	-
129	sys_delete_module	kernel/module.c	const char *	-	-	-	-
130	sys_get_kernel_syms	kernel/module.c	struct kernel_sym *	-	-	-	-
131	sys_quotactl	fs/dquot.c	int	const char *	int	caddr_t	-
132	sys_getpgid	kernel/sys.c	pid_t	-	-	-	-
133	sys_fchdir	fs/open.c	unsigned int	-	-	-	-
134	sys_bdflush	fs/buffer.c	int	long	-	-	-
135	sys_sysfs	fs/super.c	int	unsigned long	unsigned long	-	-
136	sys_personality	kernel/exec_domain.c	unsigned long	-	-	-	-
138	sys_setfsuid	kernel/sys.c	uid_t	-	-	-	-
139	sys_setfsgid	kernel/sys.c	gid_t	-	-	-	-
140	sys_llseek	fs/read_write.c	unsigned int	unsigned long	unsigned long	loff_t *	unsigned int
141	sys_getdents	fs/readdir.c	unsigned int	void *	unsigned int	-	-
142	sys_select	fs/select.c	int	fd_set *	fd_set *	fd_set *	struct timeval *
143	sys_flock	fs/locks.c	unsigned int	unsigned int	-	-	-
144	sys_msync	mm/filemap.c	unsigned long	size_t	int	-	-
145	sys_readv	fs/read_write.c	unsigned long	const struct iovec *	unsigned long	-	-
146	sys_writev	fs/read_write.c	unsigned long	const struct iovec *	unsigned long	-	-
147	sys_getsid	kernel/sys.c	pid_t	-	-	-	-
148	sys_fdatasync	fs/buffer.c	unsigned int	-	-	-	-
149	sys_sysctl	kernel/sysctl.c	struct __sysctl_ar	-	-	-	-

			gs *				
150	sys_mlock	mm/mlock.c	unsigned long	size_t	-	-	-
151	sys_munlock	mm/mlock.c	unsigned long	size_t	-	-	-
152	sys_mlockall	mm/mlock.c	int	-	-	-	-
153	sys_munlockall	mm/mlock.c	-	-	-	-	-
154	sys_sched_setparam	kernel/sched.c	pid_t	struct sched_param *	-	-	-
155	sys_sched_getparam	kernel/sched.c	pid_t	struct sched_param *	-	-	-
156	sys_sched_setscheduler	kernel/sched.c	pid_t	int	struct sched_param *	-	-
157	sys_sched_getscheduler	kernel/sched.c	pid_t	-	-	-	-
158	sys_sched_yield	kernel/sched.c	-	-	-	-	-
159	sys_sched_get_priority_max	kernel/sched.c	int	-	-	-	-
160	sys_sched_get_priority_min	kernel/sched.c	int	-	-	-	-
161	sys_sched_rr_get_interval	kernel/sched.c	pid_t	struct timespec *	-	-	-
162	sys_nanosleep	kernel/sched.c	struct timespec *	struct timespec *	-	-	-
163	sys_mremap	mm/mremap.c	unsigned long	unsigned long	unsigned long	unsigned long	-
164	sys_setresuid	kernel/sys.c	uid_t	uid_t	uid_t	-	-
165	sys_getresuid	kernel/sys.c	uid_t *	uid_t *	uid_t *	-	-
166	sys_vm86	arch/i386/kernel/vm86.c	struct vm86_struct *	-	-	-	-
167	sys_query_module	kernel/module.c	const char *	int	char *	size_t	size_t *
168	sys_poll	fs/select.c	struct pollfd *	unsigned int	long	-	-
169	sys_nfsservctl	fs/filesystems.c	int	void *	void *	-	-

170	sys_setresgid	kernel/sys.c	gid_t	gid_t	gid_t	-	-
171	sys_getresgid	kernel/sys.c	gid_t *	gid_t *	gid_t *	-	-
172	sys_prctl	kernel/sys.c	int	unsigned long	unsigned long	unsigned long	unsigned long
173	sys_rt_sigreturn	arch/i386/kernel/signal.c	unsigned long	-	-	-	-
174	sys_rt_sigaction	kernel/signal.c	int	const struct sigaction *	struct sigaction *	size_t	-
175	sys_rt_sigprocmask	kernel/signal.c	int	sigset_t *	sigset_t *	size_t	-
176	sys_rt_sigpending	kernel/signal.c	sigset_t *	size_t	-	-	-
177	sys_rt_sigtimedwait	kernel/signal.c	const sigset_t *	siginfo_t *	const struct timespec *	size_t	-
178	sys_rt_sigqueueinfo	kernel/signal.c	int	int	siginfo_t *	-	-
179	sys_rt_sigsuspend	arch/i386/kernel/signal.c	sigset_t *	size_t	-	-	-
180	sys_pread	fs/read_write.c	unsigned int	char *	size_t	loff_t	-
181	sys_pwrite	fs/read_write.c	unsigned int	const char *	size_t	loff_t	-
182	sys_chown	fs/open.c	const char *	uid_t	gid_t	-	-
183	sys_getcwd	fs/dcache.c	char *	unsigned long	-	-	-
184	sys_capget	kernel/capability.c	cap_user_header_t	cap_user_data_t	-	-	-
185	sys_capset	kernel/capability.c	cap_user_header_t	const cap_user_data_t	-	-	-
186	sys_sigaltstack	arch/i386/kernel/signal.c	const stack_t *	stack_t *	-	-	-
187	sys_sendfile	mm/filemap.c	int	int	off_t *	size_t	-
190	sys_vfork	arch/i386/kernel/process.c	struct pt_regs	-	-	-	-

Note for sys_ipc (117): this syscall takes six arguments, so it can't fit into the five registers %ebx - %edi; the last parameter (not shown) is of type 'long'. This syscall requires a special call method where a pointer is put in %ebx which points to an array containing the six arguments.

I will now explain exactly where in the kernel source that I got the information in the table above. I do this because 1) changes in the source are bound to happen, 2) you might be curious, or 3) I might've made an error.

System Call Numbers

For the numbers of the syscalls, look in arch/i386/kernel/entry.S for sys_call_table. The syscall numbers are offsets into that table. Several spots in the table are occupied by the syscall sys_ni_syscall. This is a placeholder that either replaces an obsolete syscall or reserves a spot for future syscalls.

Incidentally, the system calls are called from the function system_call in the same file; in particular, they are called with the assembly instruction 'call *SYMBOL_NAME(sys_call_table)(,%eax,4)'. The part '*SYMBOL_NAME(sys_call_table)' just gets replaced by a symbol name in sys_call_table. **SYMBOL_NAME** is a macro defined in include/linux/linkage.h, and it just replaces itself with its argument.

Typedefs

Here are the typedef declarations in the prototypes above:

atomic_t	`include/asm/atomic.h:`
	`#ifdef __SMP__`
	`typedef struct { volatile int counter; } atomic_t;`
	`#else`
	`typedef struct { int counter; } atomic_t;`
	`#endif`

caddr_t	`include/asm/posix_types.h:typedef char * __kernel_caddr_t;`
	`include/linux/types.h:typedef __kernel_caddr_t caddr_t;`

cap_user_header_t	`include/linux/capability.h:`
	`typedef struct __user_cap_header_struct {`
	`     __u32 version;`
	`     int pid;`
	`} *cap_user_header_t;`

cap_user_data_t	`include/linux/capability.h:`
	`typedef struct __user_cap_data_struct {`
	`     __u32 effective;`
	`     __u32 permitted;`
	`     __u32 inheritable;`
	`} *cap_user_data_t;`

clock_t	`include/asm/posix_types.h:typedef long __kernel_clock_t;`
	`include/linux/types.h:typedef __kernel_clock_t clock_t;`

dev_t	`include/asm/posix_types.h:typedef unsigned short __kernel_dev_t;`
	`include/linux/types.h:typedef __kernel_dev_t dev_t;`

fdset	`include/linux/posix_types.h`
	`#define __FD_SETSIZE 1024`
	`#define __NFDBITS (8 * sizeof(unsigned long))`
	`#define __FDSET_LONGS (__FD_SETSIZE/__NFDBITS)`
	`(==> __FDSET_LONGS == 32)`
	
	`typedef struct {`
	`     unsigned long fds_bits [__FDSET_LONGS];`
	`} __kernel_fd_set;`
	`include/linux/types.h:typedef __kernel_fd_set fd_set;`

gid_t	`include/asm/posix_types.h:typedef unsigned short __kernel_gid_t;`
	`include/linux/types.h:typedef __kernel_gid_t gid_t;`

__kernel_daddr_t	`include/asm/posix_types.h:typedef int __kernel_daddr_t;`

__kernel_fsid_t	include/asm/posix_types.h:
	typedef struct {
	     int __val[2];
	} __kernel_fsid_t;

__kernel_ino_t	include/asm/posix_types.h:typedef unsigned long __kernel_ino_t;
__kernel_size_t	include/asm/posix_types.h:typedef unsigned int __kernel_size_t;

loff_t	include/asm/posix_types.h:typedef long long __kernel_loff_t;
	include/linux/types.h:typedef __kernel_loff_t loff_t;

mode_t	include/asm/posix_types.h:typedef unsigned short __kernel_mode_t;
	include/linux/types.h:typedef __kernel_mode_t mode_t;

off_t	include/asm/posix_types.h:typedef long __kernel_off_t;
	include/linux/types.h:typedef __kernel_off_t off_t;

old_sigset_t	include/asm/signal.h:typedef unsigned long old_sigset_t;

pid_t	include/asm/posix_types.h:typedef int __kernel_pid_t;
	include/linux/types.h:typedef __kernel_pid_t pid_t;

__sighandler_t	include/asm/signal.h:typedef void (*__sighandler_t)(int);

siginfo_t	include/asm/siginfo.h:
	#define SI_MAX_SIZE 128
	#define SI_PAD_SIZE ((SI_MAX_SIZE/sizeof(int)) - 3)
	(==> SI_PAD_SIZE == 29)
	
	typedef struct siginfo {
	     int si_signo;
	     int si_errno;
	     int si_code;
	
	     union {
	          int _pad[SI_PAD_SIZE];
	
	          /* kill() */
	          struct {
	               pid_t _pid; /* sender's pid */
	               uid_t _uid; /* sender's uid */
	          } _kill;
	
	          /* POSIX.1b timers */
	          struct {
	               unsigned int _timer1;
	               unsigned int _timer2;
	          } _timer;
	
	          /* POSIX.1b signals */
	          struct {
	               pid_t _pid; /* sender's pid */
	               uid_t _uid; /* sender's uid */
	               sigval_t _sigval;
	          } _rt;
	
	          /* SIGCHLD */
	          struct {
	               pid_t _pid; /* which child */
	               uid_t _uid; /* sender's uid */
	               int _status; /* exit code */
	               clock_t _utime;
	               clock_t _stime;
	          } _sigchld;
	
	          /* SIGILL, SIGFPE, SIGSEGV, SIGBUS */

	struct {
	void *_addr; /* faulting insn/memory ref. */
	} _sigfault;
	
	/* SIGPOLL */
	struct {
	int _band; /* POLL_IN, POLL_OUT, POLL_MSG */
	int _fd;
	} _sigpoll;
	} _sifields;
	} siginfo_t;

sigset_t	include/asm/signal.h:typedef unsigned long sigset_t;

size_t	include/asm/posix_types.h:typedef unsigned int __kernel_size_t;
	include/linux/types.h:typedef __kernel_size_t size_t;

ssize_t	include/asm/posix_types.h:typedef int __kernel_ssize_t;
	include/linux/types.h:typedef __kernel_ssize_t ssize_t;

stack_t	include/asm/signal.h:
	typedef struct sigaltstack {
	void *ss_sp;
	int ss_flags;
	size_t ss_size;
	} stack_t;

suseconds_t	include/asm/posix_types.h:typedef long __kernel_suseconds_t;
	include/linux/types.h:typedef __kernel_suseconds_t suseconds_t;

time_t	include/asm/posix_types.h:typedef long __kernel_time_t;
	include/linux/types.h:typedef __kernel_time_t time_t;

uid_t	include/asm/posix_types.h:typedef unsigned short __kernel_uid_t;
	include/linux/types.h:typedef __kernel_uid_t uid_t;

uint	include/linux/types.h:typedef unsigned int uint;
__u32	include/asm/types.h:typedef unsigned int __u32;

Struct Declarations

Here are the struct declarations for the table at the top:

exception_table_entry	include/linux/module.h:
	struct exception_table_entry {
	unsigned long insn, fixup;
	};

iovec	include/linux/uio.h:
	struct iovec {
	void *iov_base;
	__kernel_size_t iov_len; };

itimerval	include/linux/time.h:
	struct itimerval {
	struct timeval it_interval; /* timer interval */
	struct timeval it_value; /* current value */
	};

kernel_sym	include/linux/module.h:
	struct kernel_sym {
	unsigned long value;
	char name[60];
	};

mmap_arg_struct	arch/i386/kernel/sys_i386.c:

	struct mmap_arg_struct {
	    unsigned long addr;
	    unsigned long len;
	    unsigned long prot;
	    unsigned long flags;
	    unsigned long fd;
	    unsigned long offset;
	};

module	include/linux/module.h:
	struct module {
	    unsigned long size_of_struct; /* sizeof(module) */
	    struct module *next;
	    const char *name;
	    unsigned long size;
	    union {
	        atomic_t usecount;
	        long pad;
	    } uc;
	    unsigned long flags; /* AUTOCLEAN et al */
	    unsigned nsyms;
	    unsigned ndeps;
	
	    struct module_symbol *syms;
	    struct module_ref *deps;
	    struct module_ref *refs;
	    int (*init)(void);
	    void (*cleanup)(void);
	    const struct exception_table_entry *ex_table_start;
	    const struct exception_table_entry *ex_table_end;
	/* Members past this point are extensions to the basic
	module support and are optional. Use mod_opt_member()
	to examine them. */
	    const struct module_persist *persist_start;
	    const struct module_persist *persist_end;
	    int (*can_unload)(void);
	};

module_persist	include/linux/module.h: struct module_persist; /* yes, it's empty */

module_ref	include/linux/module.h:
	struct module_ref {
	    struct module *dep; /* "parent" pointer */
	    struct module *ref; /* "child" pointer */
	    struct module_ref *next_ref;
	};

module_symbol	include/linux/module.h:
	struct module_symbol {
	    unsigned long value;
	    const char *name;
	};

new_utsname	include/linux/utsname.h:
	struct new_utsname {
	    char sysname[65];
	    char nodename[65];
	    char release[65];
	    char version[65];
	    char machine[65];
	    char domainname[65];
	};

__old_kernel_stat	include/asm/stat.h:
	struct __old_kernel_stat {
	unsigned short st_dev;
	unsigned short st_ino;
	unsigned short st_mode;
	unsigned short st_nlink;
	unsigned short st_uid;
	unsigned short st_gid;
	unsigned short st_rdev;
	unsigned long st_size;
	unsigned long st_atime;
	unsigned long st_mtime;
	unsigned long st_ctime;
	};

oldold_utsname	include/linux/utsname.h:
	struct oldold_utsname {
	char sysname[9];
	char nodename[9];
	char release[9];
	char version[9];
	char machine[9];
	};

old_sigaction	include/asm/signal.h:
	struct old_sigaction {
	__sighandler_t sa_handler;
	old_sigset_t sa_mask;
	unsigned long sa_flags;
	void (*sa_restorer)(void);
	};

old_utsname	include/linux/utsname.h:
	struct old_utsname {
	char sysname[65];
	char nodename[65];
	char release[65];
	char version[65];
	char machine[65];
	};

pollfd	include/asm/poll.h:
	struct pollfd {
	int fd;
	short events;
	short revents;
	};

pt_regs	include/asm/ptrace.h:
	struct pt_regs {
	long ebx;
	long ecx;
	long edx;
	long esi;
	long edi;
	long ebp;
	long eax;
	int xds;
	int xes;
	long orig_eax;
	long eip;
	int xcs;
	long eflags;
	long esp;

	int xss;
	};

revectored_struct	include/asm/vm86.h:
	struct revectored_struct {
	unsigned long __map[8];
	};

rlimit	include/linux/resource.h:
	struct rlimit {
	long rlim_cur;
	long rlim_max;
	};

rusage	include/linux/resource.h:
	struct rusage {
	struct timeval ru_utime; /* user time used */
	struct timeval ru_stime; /* system time used */
	long ru_maxrss; /* maximum resident set size */
	long ru_ixrss; /* integral shared memory size */
	long ru_idrss; /* integral unshared data size */
	long ru_isrss; /* integral unshared stack size */
	long ru_minflt; /* page reclaims */
	long ru_majflt; /* page faults */
	long ru_nswap; /* swaps */
	long ru_inblock; /* block input operations */
	long ru_oublock; /* block output operations */
	long ru_msgsnd; /* messages sent */
	long ru_msgrcv; /* messages received */
	long ru_nsignals; /* signals received */
	long ru_nvcsw; /* voluntary context switches */
	long ru_nivcsw; /* involuntary " */
	};

sched_param	include/linux/sched.h:
	struct sched_param {
	int sched_priority;
	};

sel_arg_struct	arch/i386/kernel/sys_i386.c:
	struct sel_arg_struct {
	unsigned long n;
	fd_set *inp, *outp, *exp;
	struct timeval *tvp;
	};

sigaction	include/asm/signal.h:
	struct sigaction {
	__sighandler_t sa_handler;
	unsigned long sa_flags;
	void (*sa_restorer)(void);
	sigset_t sa_mask; /* mask last for extensibility */
	};

stat	include/asm/stat.h:
	struct stat {
	unsigned short st_dev;
	unsigned short __pad1;
	unsigned long st_ino;
	unsigned short st_mode;
	unsigned short st_nlink;
	unsigned short st_uid;
	unsigned short st_gid;
	unsigned short st_rdev;
	unsigned short __pad2;

	unsigned long st_size;
	unsigned long st_blksize;
	unsigned long st_blocks;
	unsigned long st_atime;
	unsigned long __unused1;
	unsigned long st_mtime;
	unsigned long __unused2;
	unsigned long st_ctime;
	unsigned long __unused3;
	unsigned long __unused4;
	unsigned long __unused5;

statfs	include/asm/statfs.h:
	struct statfs {
	long f_type;
	long f_bsize;
	long f_blocks;
	long f_bfree;
	long f_bavail;
	long f_files;
	long f_ffree;
	__kernel_fsid_t f_fsid;
	long f_namelen;
	long f_spare[6];
	};

__sysctl_args	include/linux/sysctl.h
	struct __sysctl_args {
	int *name;
	int nlen;
	void *oldval;
	size_t *oldlenp;
	void *newval;
	size_t newlen;
	unsigned long __unused[4];
	};

sysinfo	include/linux/kernel.h:
	struct sysinfo {
	long uptime; /* Seconds since boot */
	unsigned long loads[3]; /* 1, 5, and 15 minute load averages */
	unsigned long totalram; /* Total usable main memory size */
	unsigned long freeram; /* Available memory size */
	unsigned long sharedram; /* Amount of shared memory */
	unsigned long bufferram; /* Memory used by buffers */
	unsigned long totalswap; /* Total swap space size */
	unsigned long freeswap; /* swap space still available */
	unsigned short procs; /* Number of current processes */
	char _f[22]; /* Pads structure to 64 bytes */
	};

timex	include/linux/timex.h: struct timex { unsigned int modes; /* mode selector */ long offset; /* time offset (usec) */ long freq; /* frequency offset (scaled ppm) */ long maxerror; /* maximum error (usec) */ long esterror; /* estimated error (usec) */ int status; /* clock command/status */ long constant; /* pll time constant */ long precision; /* clock precision (usec) (read only) */ long tolerance; /* clock frequency tolerance (ppm) * (read only) */

	struct timeval time; /* (read only) */ long tick; /* (modified) usecs between clock ticks */ long ppsfreq; /* pps frequency (scaled ppm) (ro) */ long jitter; /* pps jitter (us) (ro) */ int shift; /* interval duration (s) (shift) (ro) */ long stabil; /* pps stability (scaled ppm) (ro) */ long jitcnt; /* jitter limit exceeded (ro) */ long calcnt; /* calibration intervals (ro) */ long errcnt; /* calibration errors (ro) */ long stbcnt; /* stability limit exceeded (ro) */ int :32; int :32; int :32; int :32; int :32; int :32; int :32; int :32; int :32; int :32; int :32; int :32; };

timespec	include/linux/time.h:
	struct timespec {
	time_t tv_sec; /* seconds */
	long tv_nsec; /* nanoseconds */
	};

timeval	include/linux/time.h:
	struct timeval {
	time_t tv_sec; /* seconds */
	suseconds_t tv_usec; /* microseconds */
	};

timezone	include/linux/time.h:
	struct timezone {
	int tz_minuteswest; /* minutes west of Greenwich */
	int tz_dsttime; /* type of dst correction */
	};

tms	include/linux/times.h
	struct tms {
	clock_t tms_utime;
	clock_t tms_stime;
	clock_t tms_cutime;
	clock_t tms_cstime;
	};

ustat	include/linux/types.h:
	struct ustat {
	__kernel_daddr_t f_tfree;
	__kernel_ino_t f_tinode;
	char f_fname[6];
	char f_fpack[6];
	};

utimbuf	include/linux/utime.h:
	struct utimbuf {
	time_t actime;
	time_t modtime;
	};

vm86plus_info_struct	include/asm/vm86.h:
	struct vm86plus_info_struct {
	unsigned long force_return_for_pic:1;
	unsigned long vm86dbg_active:1;
	unsigned long vm86dbg_TFpendig:1;
	unsigned long unused:28;
	unsigned long is_vm86pus:1;
	unsigned char vm86dbg_intxxtab[32];
	};

vm86plus_struct	include/asm/vm86.h:
	struct vm86plus_struct {
	    struct vm86_regs regs;
	    unsigned long flags;
	    unsigned long screen_bitmap;
	    unsigned long cpu_type;
	    struct revectored_struct int_revectored;
	    struct revectored_struct int21_revectored;
	    struct vm86plus_info_struct vm86plus;
	};

vm86_regs	include/asm/vm86.h:
	struct vm86_regs {
	/* normal regs, with special meaning for the segment descriptors.. */
	    long ebx;
	    long ecx;
	    long edx;
	    long esi;
	    long edi;
	    long ebp;
	    long eax;
	    long __null_ds;
	    long __null_es;
	    long __null_fs;
	    long __null_gs;
	    long orig_eax;
	    long eip;
	    unsigned short cs, __csh;
	    long eflags;
	    long esp;
	    unsigned short ss, __ssh;
	/* these are specific to v86 mode: */
	    unsigned short es, __esh;
	    unsigned short ds, __dsh;
	    unsigned short fs, __fsh;
	    unsigned short gs, __gsh;
	};

vm86_struct	include/asm/vm86.h:
	struct vm86_struct {
	    struct vm86_regs regs;
	    unsigned long flags;
	    unsigned long screen_bitmap;
	    unsigned long cpu_type;
	    struct revectored_struct int_revectored;
	    struct revectored_struct int21_revectored;
	};

END INCLUDED FILE