RC24725 (W0901-048) January 14, 2009
Computer Science

IBM Research Report

Isolation Mechanisms for Commodity Applications and Platforms

Wietse Venema
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Isolation mechanisms for commodity applications and platforms

Wietse Venema,
IBM T.J.Watson Research center,
Hawthorme, New York, USA.

Web clients have evolved into user−level operating systems. They run applications from friendly and hostile providers side by side, often within a single process address space. This presents a poor match for legacy operating systems that assign application privileges based on local user identities, and that implement isolation domains with process granularity. This report reviews a number of techniques for contemporary systems that can partition an application into trusted and untrusted parts. These techniques can enforce different policies for different applications that run on behalf of the same user, even when the applications become corrupted.

1 Introduction

Web browsers have become a universal platform to execute applications such as email, other social interaction, multimedia, and commerce. Web browsers provide OS−independent execution environments, and threaten existing monopolies on application suites. Unfortunately, web browsers also offer plenty opportunity for penetration of the client platform.

- Web browsers are large monolithic programs built from millions of lines of source code. Most of this code is written in unsafe implementation languages such as C or C++. From January 2008 through November 2008, the Mozilla foundation released 53 advisories for the Firefox 2.0 browser, of which 23 were rated critical, meaning that they could "be used to run attacker code and install software, requiring no user interaction beyond normal browsing" [Mozilla 2008].

- Third−party extensions are written in unsafe implementation languages, and are often distributed as closed−source binary code. This code is linked into the browser process address space via the Windows dynamic−link library (DLL) mechanism or via the Netscape plug−in API (NPAPI). Unpatched vulnerabilities in third−party extensions are a major contributor to web client compromises [Provos 2008].

- Active content such as Java applets, JavaScript and VBScript code is downloaded on request of web pages, and executes inside the web browser. On Windows systems, such code may directly attack code in any local DLL file that is marked as safe for scripting.

- Different web applications execute side by side in the same browser. Often these applications run inside a single process address space and rely entirely

on the browser to enforce isolation.

- Authorization models of legacy operating systems assign privileges based on a local user's identity. The models do not distinguish between web applications from different parties that run on behalf of the same local user.

- Mashups aggregate content from multiple providers into a single web page, and introduce further challenges to isolating applications from different providers [Wang 2007; de Keukelaere 2007].

Web browsers have evolved from document rendering engines into multi−user operating systems that need to isolate untrusted applications from each other and from the local system. To make this task even more challenging, browsers must implement their isolation policies entirely in user−land code, without assistance from the underlying operating system.

This report reviews a number of isolation mechanisms that are available for contemporary operating systems. The mechanisms may be used to partition web browsers into subsystems that execute web applications in isolation from each other and from the host platform. Covert channels are not discussed in this report; the primary focus is on preventing web applications from corrupting other applications or the host platform itself.

The organization of this report is as follows: after the remainder of the introduction, section 2 gives an overview of sandboxing mechanisms. Sections 3−6 follow with more detailed discussions. Section 7 looks at solutions that were developed for other system architectures. Section 8 discusses findings, and section 9 concludes this report.

1.1 Current status

Recent system−level efforts partition the browser, and run the partitions under control of a trusted monitor that enforces policies as it manages communication between partitions, with the network, with local storage and with the user interface.

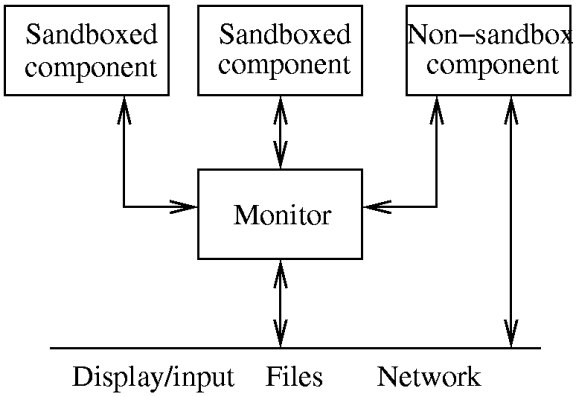

Figure 1. Basic partitioned web browser architecture.

Specific implementations differ in the degree of isolation, and range from the Tahoma browser with one virtual machine per website [Cox 2006]; the OP browser with sandboxed processes per website, per browser subsystem, and per browser extension [Grier 2008]; to Google's Chromium browser with sandboxed processes per website and non−sandboxed processes per browser extension [Barth 2008]. Others have proposed an approach that is explicitly inspired by microkernel−based operating systems [Singh 2008]. This report will focus on the underlying mechanisms that can be used in such designs, and the limitations of those mechanisms.

At the language level, boundaries between applications have been pioneered recently by rewriting HTML and JavaScript on the fly [Reis 2006], and by proposing HTML tags that introduce the concept of a sandbox that limits the resources available to web applications [Wang 2007]. These approaches will not be discussed in this report.

2 Sandboxing

All mechanisms discussed in this report execute code inside an isolated environment, also called a sandbox. Sandboxes come in different sizes: a sandbox may be a small portion of a process address space, it may contain an entire process or group of related processes, or it may even contain a complete operating system instance. Code that executes inside a sandbox has no direct access to data or code outside; however, non−sandboxed code may have access to data or code inside a sandbox.

When sandboxed code wants to access an external resource such as a file or other application, control switches to a trusted sandbox monitor that either denies the request or forwards the request on behalf of the sandboxed code. Effectively, the sandbox monitor interposes on all communication between the sandboxed code and the world outside, and enforces an access policy. Software that implements this role is a special case of a reference monitor.

Figure 2. Basic sandbox architecture. The reader beware: not all sandboxes discussed in this report actually implement this model.

To give the reader a quick idea of the types of technology and their applicability, we present a brief summary of isolation techniques. Many of these techniques will be discussed in more detail later in this report.

2.1 Managed code

Some programming systems compile application program statements into executable code for a virtual machine. These pseudo−machine instructions execute under control by software. That software not only manages application resources, but also provides complete mediation between the application and its host environment. Contemporary examples of managed−code environments are the Java Virtual Machine (JVM) which executes Java bytecodes [SUN Java 2008], and Microsoft's Common Language Runtime (CLR) which executes Common Intermediate Language (CIL) bytecodes

produced by C#, VB.NET, and other compilers [Microsoft CLR 2008]. According to an older estimate, typical JVMs contain about 10^5 lines of trusted code [Appel 2002].

Compared to applications that are implemented in unsafe languages such as C or C++, managed−code applications are relatively immune to memory and control corruption problems. As long as managed code does not manipulate memory addresses, corruption problems are limited to the unmanaged code that implements virtual machine itself and the interface to the host environment. The best−performing managed−code environments compile pseudo−machine instructions into native code, and have a runtime overhead of the order of 10% compared to unmanaged code.

The obvious limitation of managed−code systems is that they support only applications that are implemented in the supported programming languages. However, a large body of code is written in these languages, so they deserve serious consideration.

2.2 In−line reference monitor

The second option is to integrate the access policy within the application code itself, such that the policy is enforced even when the application code has bugs. This is the goal of software isolation (XFI), discussed in section 3.

In brief, the idea of software isolation is to enforce memory access policies in software similar to a hardware memory−management unit. This is usually implemented by performing a static analysis of binary code and inserting run−time checks with each load, store or jump instruction that cannot be verified statically. The primary goal is to confine memory access to specific memory address ranges; often, this also requires some form of control−flow enforcement.

All interactions between the isolated application and its environment are mediated by trusted code in a separate memory region. The run−time checks may be inserted at compile time by a dedicated compiler stage, at program startup time, or on−demand at program runtime. With a run−time overhead of the order of 10%, the running time of software−isolated code is comparable to that of managed−code virtual machines with a just−in−time code generator.

2.3 System call sandbox

With modern systems, privileged operating system code handles all interactions between a process and its environment. Even communication via shared memory or memory−mapped files requires systems calls to set up the memory mappings. System call interposition is a popular approach to monitor application activity and to augment operating system policies, and is discussed in section 4.

System call sandboxes are typically implemented by invoking monitoring code before and after execution of each system call of interest. The monitoring code examines system call arguments and application state, and may restrict, modify or audit application activity. However, this approach also introduces opportunities for race conditions. Implementors have devised a number of solutions or workarounds, not all

of which work.

2.4 Hardware memory isolation

Hardware memory management is normally under exclusive control of an operating system. However, modern ix86−based versions of Linux, BSD, Mac OS X, and Microsoft Windows implement system calls that provide applications with limited control over memory segments (contiguous memory ranges with their own protection settings).

By using memory segments as isolation domains, it becomes possible to safely execute untrusted code without the overhead of address checking instructions, and to switch between isolation domains without the cost of a hardware context switch. This approach requires minor application code modifications to prevent untrusted applications from executing unsafe instructions, such as system calls, instructions that manipulate memory segments, or instructions that can jump into the middle of variable−length instructions. These modifications can be applied on the fly at runtime, or in advance at application compile time.

Hardware memory isolation is discussed in section 5.

2.5 Resource virtualization

Interposition has uses beyond application confinement. It may also be used to virtualize an application's view of its environment. With some architectures, the interposition mechanism hides resources that aren't supposed to be visible from inside a sandbox. From a sandboxed application's point of view, such resources simply do not exist. This is the basis for BSD jails and Solaris zones. With other architectures, the interposition mechanism translates between virtual names inside the sandbox and physical names outside; this is the basis for API virtualization and hardware virtualization. Resource virtualization is the subject of section 6.

After this high−level overview we now discuss isolation mechanisms in more detail.

3 In−line reference monitors

Hardware memory protection comes at a price. Besides providing isolation between memory domains, it also increases the cost of communication between domains. The Singularity project has measured the cost of virtual memory, hardware isolation, and privilege levels on ix86 hardware. They find a factor of 3 slowdown in microbenchmarks with frequent calls from unprivileged code into a monolithic kernel, compared to calls between real−mode code that executes with "ring 0" privileges [Aiken 2006].

To avoid the cost of hardware cross−domain communication, many operating systems and applications load native−code extensions into their address spaces without any form of isolation. The resulting gain in performance comes at the cost of stability. According to older studies, Windows/XP device driver bugs are the cause of 85% of system crashes [Switch 2003], and Linux device driver bugs have up to seven

times the frequency of other kernel bugs [Chou 2001]. At the application level, bugs in third−party extensions are a frequent cause of web browser failures.

Software fault isolation (SFI) avoids the cost of hardware cross−domain communication by implementing memory domain boundaries in software. It enforces memory isolation by inserting run−time checks before load, store and jump instructions. Just like hardware isolation, SFI only confines memory access to specific memory ranges. It does not stop an attacker from overwriting arbitrary portions of memory within a range, or from changing the flow of program execution.

XFI stands for SFI combined with control−flow integrity (CFI). CFI assures that program execution follows a pre−determined control−flow graph. This has obvious benefits for stopping control hijacking via memory corruption attacks (for example, overwriting a function return address on the runtime stack). CFI also provides opportunities to make SFI more efficient. For example, SFI memory safety checks may be moved outside loops.

Software isolation relies on correct instrumentation of the untrusted code, plus correct implementation of the trusted code that implements the interface between the sandboxed code and its environment. The program that instruments the application does not need to be trusted, as long as it is possible to statically verify that the instrumented code is safe.

The next sections discuss SFI and CFI in more detail, including specific implementations.

3.1 SFI − Software fault isolation

With SFI, a compiler or loader instruments code, such that all memory accesses are confined to specific code and data address ranges, which are called sandboxes. The idea is to do a static analysis of load, store and jump instructions, and to insert run−time checks whenever the target of an instruction cannot be verified statically. Since the instrumentation is applied at the machine instruction level, it is independent of higher−level implementation languages.

Separation of code and data sandboxes prevents instrumented code from modifying its own instructions. In the absence of control−flow integrity (to be discussed below), the code/data sandbox separation also prevents extensions from jumping into data and thus executing arbitrary instructions.

Software that runs inside a software fault isolation sandbox must not be allowed to make direct system calls; this would allow errant or malicious code to manipulate resources, such as memory or file handles, that belong to code outside its sandbox. Instead, sandboxed code needs to invoke trusted code that mediates system calls, calls across software isolation domains, and calls into non−sandboxed code. The trusted code enforces the sandbox policy on function call arguments and results, and properly saves and restores processor registers across calls.

The practical cost of software fault isolation is a trade−off between complexity,

performance, safety, and ease of debugging. Some implementations force code and data addresses to "wrap around" within their respective sandboxes; this simplifies implementation, but complicates debugging. Some implementations do not sandbox load instructions; this can be acceptable in environments without secrets or destructive read operations. Other implementations allow function call returns and other jumps to any address within a code sandbox; this simplifies the code sandbox implementation, but increases the cost of the sandbox, especially with processors that execute non−aligned instructions, or that have variable−length instructions. With such processors, jumping into the middle of a multi−byte instruction could result in the execution of instructions that violate sandbox policy.

3.1.1 Classic SFI

SFI [Wahbe 1993] has two primary motivations: to confine less trusted guest software that runs in the address space of more trusted host software, and to implement low−cost communication between protection domains. Security is not one of the primary goals.

The idea is to introduce software fault domains as subsets of hardware protection domains. Each software fault domain is allowed to access only its own code and data address ranges. The implementation uses a special compiler to instrument "unsafe" loads, stores and jumps. This paper introduces the address "wrap around" technique to confine data and memory address within their respective sandboxes.

The implementation by Wahbe et al. targets the RISC−based MIPS and ALPHA processors, and reserves 4 to 5 processor registers. The run−time overhead of a jump/store sandbox is 5% in macro benchmarks. The run−time overhead increases to 20% when load instructions are sandboxed, too.

```
; opcode    dest, source
load        data-reg, address
and         data-reg, data-mask
or          data-reg, data-prefix
store       [data-reg], value
```

Figure 3. Traditional SFI sandbox address wrap−around technique. By reserving register *data−reg* for address calculations, the SFI implementation guarantees that *data−reg* always contains an address within the data sandbox, even when execution jumps directly into the store instruction.

A drawback of this approach is that it is non−trivial to implement on processors with a limited number of registers, or with variable−length instructions, such as the ix86 processor family.

3.1.2 PittSFIeld − SFI for CISC processors

The PittSFIeld system [McCamant 2006] overcomes the problems with the ix86's limited registers and variable−length instructions. The implementation consists of two parts: an untrusted compiler stage, and a trusted load−time verifier of approximately

800 lines of C code.

The compiler stage breaks up the assembly−level instruction stream into blocks of up to 16 bytes, and requires that jump destination addresses are multiples of 16. Each block contains both the memory access check and the instruction that it protects. Short instruction blocks are padded with NOP instructions. As the result of clever optimizations, PittSFIeld requires only one reserved register, for the address prefix of the separate code and data address "wrap around" sandboxes.

Figure 4. Example of machine instructions aligned to 16−byte blocks. NOP padding is shown in white. Call instructions are placed at the end of 16−byte blocks, so that return instructions will jump to the start of the next block.

The run−time overhead of a jump/store sandbox is 21% for SPECint2000 benchmarks; program sizes increase by 70% on average. Roughly half the run−time overhead is due to the padding with NOP instructions. These instructions not only take time to execute, but also decrease cache efficiency. In a macro performance comparison with VXA archive decompressors [Ford 2005], the run−time overhead is 2.8% with VX32's ix86 segment−based sandboxing, and 28% with PittSFIeld's jump/store sandboxing. VX32 is introduced in section 5.2.

As described, PittSFIeld does not sandbox memory load instructions. This is not a fundamental limitation. Adding load sandboxing would, however, slow down execution further. PittSFIeld would have to be re−architected if it were to be implemented as a trusted program loader instead of a compiler stage plus load−time verifier.

Erlingsson et al. [Erlingsson 2006] observe that the PittSFIeld implementation is subject to TOCTOU race conditions when it dereferences return addresses, or when it loads processor flags from memory. These are not fundamental limitations.

3.1.3 Other SFI approaches

MiSFIT implements SFI for C++ program extensions on ix86 processors [Small 1998]. It is implemented as a compiler stage that transforms assembly language. MiSFIT provides a limited form of control−flow integrity: virtual function pointers must match a table of function entry points, and a shadow stack contains copies of function return addresses and callee−saved registers. The shadow stack is protected by placing it outside the data sandbox.

There is no explicit code verification step: an extension is either distributed as source code and compiled before installation, or it is distributed as digitally−signed sandboxed code. The runtime overhead for a collection of operating extensions [Small 1996] ranges from 11−22% for a call/store sandbox to 144% for a

call/store/load sandbox. Selected SPECint 1992 and SPECint 1995 benchmarks show a run−time overhead of 30−60% (call/store sandbox) and 70−90% (call/store/load sandbox). As pointed out in [McCamant 2006], MiSFIT allows the call stack to overwrite non−stack memory.

3.2 CFI − Control−flow integrity

CFI provides assurance that program execution follows a pre−determined control−flow graph [Abadi 2005a]. As with SFI, the assurance does not rely on data memory integrity. The implementation uses a combination of intra−procedural static analysis, plus run−time checks to ensure that jumps (including calls and returns) follow the control−flow graph, even when function pointers are used. The run−time checks compare numeric labels which are inserted at both jump instructions and jump targets. A more formal background for CFI is given in [Abadi 2005b], based on a generic RISC architecture, though the CFI implementation is for ix86.

In practice, one function may be called from different locations, and one location may call different functions through a function pointer. Because of this, the same numeric label will appear at multiple jump destinations. To ensure that function calls return to their most recent calling sites, CFI uses a shadow call stack. The shadow stack is protected with ix86 segmentation, and eliminates the need to check function call return addresses.

SPEC2000 macro performance benchmarks of CFI without protected call stack show an average memory size overhead of 8%, and an average run−time overhead of 16%. CFI with protected shadow call stack has a run−time overhead of 21%; no memory size overhead is given, but the difference will not matter in real life.

CFI has benefits for SFI implementations. For example, SFI memory address checks can be moved outside a loop, because CFI ensures that a loop can be entered only via one path. Comparison with a kernel extension benchmark [Small 1996] shows that CFI+SFI call/load/store protection has 10x less run−time overhead than MiSFIT [Small 1998]. The comparisons with PittSFIeld appear to be invalid because PittSFIeld does not sandbox memory load instructions [McCamant 2006].

3.3 XFI − CFI plus SFI and more

XFI [Erlingsson 2006] generalizes earlier work on CFI by the same group. It introduces a more flexible data memory model than the SFI address "wrap around" approach; one sandbox can have multiple code and data memory ranges, and different memory ranges can have different access permissions. This flexibility simplifies cross−domain communication, but increases run−time overhead and program size.

The XFI implementation uses two stacks. The execution stack is for variables and call/return linkage information, and is not addressable with pointers; variables are accessed with constant offsets relative to the stack. As with CFI, the protected stack eliminates the need to check return addresses. The allocation stack is for items that may be accessed via pointers, and can therefore become corrupted just like the heap.

XFI has been used to implement kernel drivers, codecs and DLLs for Windows on ix86 hardware. The run−time overhead of an earlier benchmark [Small 1996] is comparable to MiSFIT [Small 1998]. In contrast, the CFI paper from the same group claimed a factor 10 less overhead than MiSFIT. The XFI memory size overhead is of the order of 100%, comparable of that of SFI for CISC [McCamant 2006].

3.4 WIT − Write integrity tests

Sofar, software isolation has ensured that memory writes happen only in designated memory ranges, without regard for the boundaries between individual data objects in those ranges. This leaves software exposed to non−control−data attacks that corrupt memory by overwriting user credentials, configuration data, or other decision−making information [Chen 2005]. To block memory corruption attacks, WIT [Akritidis 2008] uses a combination of write integrity and control−flow integrity.

Based on inter−procedural static analysis of C or C++ source code, memory write instructions are considered safe when they cannot violate data or control−flow integrity; data objects are considered safe only when all their writes are safe. Unsafe write instructions and unsafe data objects are given colors; runtime checks ensure that each unsafe write instruction matches the color of the written−to memory location. Coloring is also used to match indirect jumps with jump targets. To compensate for limited precision of static analysis, guard regions with a reserved color are inserted around unsafe data objects; these guard regions are used to detect buffer overflow or underflow errors that have a sequential access pattern.

The average runtime overhead for a number of SPEC CPU 2000 benchmarks is about 10%, and the memory size overhead is about 13%. The memory overhead is mostly due to the color map which uses one byte for every 8 bytes of memory. Due to the 8−byte granularity of protection, unsafe memory objects need to be padded to align their boundaries with the color map.

WIT uses wrappers around system calls to ensure that arguments have the colors that the wrapper is allowed to write, and it uses a WIT−instrumented version of the (Windows) C library.

Besides the limitation that it requires source code, WIT has several other limitations. Library code is compiled separately from the application, and treats all caller's unsafe colors as if they were the same. In principle this could be overcome with a wrapper technique as used with system calls. For ABI compatibility reasons, WIT does not insert guard regions between member fields within a data structure. Combined with WIT's use of one color for all members within a data structure, this leaves opportunities for non−control−data attacks. WIT's color map does, however, limit the damage that may result from data pointer or function pointer corruption.

4 System−call sandboxes

With modern operating systems, system calls handle all interactions between a process and its environment. Even communication via shared memory or

memory−mapped files requires systems calls to set up the memory mappings. System call interposition is therefore a popular approach to monitor application activity and to augment operating system policies.

4.1 UNIX System−call wrappers

Many UNIX system call interposition implementations use the same basic architecture; they are implemented as a wrapper around the generic process/kernel system call entry point. The general course of events is as follows:

1. A process makes a system call and passes control to the kernel.
2. The system call wrapper passes control to a system call monitor which can inspect the system call arguments and application memory. The monitor may deny the call and return control to the monitored process, or it may modify the system call's type or arguments.
3. The system call wrapper passes control to the kernel code that implements the system call.
4. The system call wrapper passes control to the system call monitor which can inspect system call arguments, application memory, and results.
5. The system call wrapper passes control to the monitored process.

With early implementations, the system call monitor runs as a user−level process; examples are Janus [Goldberg 1996], MAPbox [Acharya 2000], and [Jaing 2000]. Janus's footprint is about 2100 lines of code. These implementations use the ptrace() process debugging interface. This approach has an obvious handicap: the monitoring process has no access to in−kernel information such as the monitored process's current directory or its effective privileges. Instead, the monitor has to duplicate kernel behavior as it witnesses system calls.

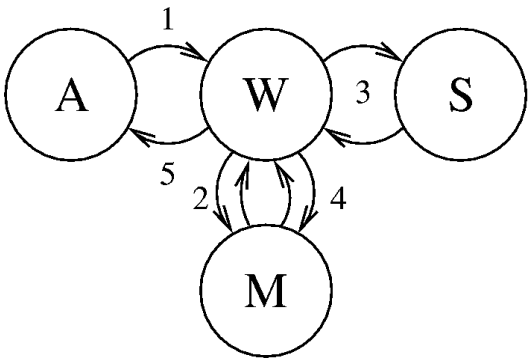

Figure 5. Control flow between application *A*, wrapper *W*, monitor *M* and system call *S*. The numbers in the figure correspond to the discussion in the text.

Later wrapper−based system call monitors are implemented in the kernel, typically as a kernel module, and sometimes with a user−level decision−making process; examples are GSWTK [Fraser 1999], BlueBoX [Chari 2002], Systrace [Provos 2003] and Ostia [Garfinkel, 2004]. Ostia's footprint is about 200 lines in the kernel and 3200 lines in the monitor. A kernel−based implementation has the potential for better performance, some of which will be lost with callouts to user−land helpers.

Unfortunately, the system call wrapper architecture is prone to race conditions, because monitors and system calls make their decisions independently [Garfinkel 2003; Watson 2007]. Race conditions may be external to the monitored process: for example, monitors and system calls resolve application pathnames at different points in time. Conspiring threads or processes can exploit this by playing games in the file system with hardlinks or symlinks.

Race conditions may also be internal to the monitored process: this happens when monitors and system calls read from user−land memory at different points in time. Conspiring threads or processes can exploit this behavior, for example by changing the contents of a pathname or network address data structure in shared memory. Memory races can be won easily even on uniprocessors, by placing data partially on a non−resident memory page [Watson 2007].

The first problem, resolving pathnames at different points in time, can be solved only with architectural changes. The solution is to invoke the system call monitor after user−land information is copied into the kernel and after pathnames etc. have been resolved to the underlying system objects. Of course, this solution eliminates the next problem, too.

The second problem, reading user−land memory at different points in time, can be worked around by using a protected copy of that information. The initial Systrace implementation saves normalized pathname arguments to a look−aside buffer in kernel memory [Provos 2003]; Ostia uses system call delegation, where a trusted monitor process receives a copy of the arguments and makes the system call on behalf of the monitored process [Garfinkel 2004].

4.2 Google Chromium sandbox

The Google Chromium browser executes processes in low−privilege sandboxes [Chromium Sandbox 2008], where each sandbox delegates Windows API calls to a monitoring process (the Windows API is implemented by libraries that make undocumented kernel calls). The initial Chrome version does not sandbox browser extensions.

In 2007, Google acquired GreenBorder's patented sandboxing technology which was used in a Windows web browser security product (e.g., [Erlingsson 2007]). Reportedly, several members of the Chromium developer team are former GreenBorder employees [InformationWeek 2008; Kennedy, 2008].

To prevent processes from sidestepping the reference monitor, the Chromium sandbox revokes privileges during process initialization. This is achieved either with Windows Vista integrity levels, or with Windows XP security tokens. The latter will not revoke access to so−called "zero−security" resources such as FAT/FAT32 file systems or TCP/IP networking [Chromium Sandbox 2008].

Windows versions before Vista do not enforce security on communication between applications that share the same desktop. For this reason, Chromium sandboxes share an alternate desktop, and thus can't be used for multimedia applications. Windows

Vista and later enforce integrity−level checks that can isolate less trusted applications from other applications that share the same desktop.

4.3 System−call domains

The system call monitors in the previous sections make no structural changes to the kernel. This limits them to relatively simple policies. The approaches in this section introduce kernel mechanisms that implement a notion of domains, where different domains can enforce different policies on processes that share the same domain.

TRON [Berman 1995] executes a process in a domain with reduced file system access permissions. A process enters a TRON domain with the tron_fork() system call; child processes inherit the domain attribute from their parent. The tron_fork() system call may be used recursively, but only if the child domain's privileges don't exceed the parent domains's privileges. TRON also supports transitive delegation of permissions from one process to another domain, and supports non−transitive revocation of delegated permissions. TRON is implemented for Ultrix, a BSD UNIX descendant.

Peterson et al. describe a more extensive design that also covers non−file resources, and evaluate a partial implementation for Linux [Peterson 2003]. In this work, system−call domains may nest, and effective permissions are computed by intersection with parent domain permissions. As with TRON, a sandbox−aware process may create a nested domain to delegate a subset of its permissions. Run−time performance depends on the number of nested domains that need to be checked, and on whether or not decisions are delegated to a user−land process.

5 Hardware isolation

Hardware memory management is normally under exclusive control of an operating system. There are two major approaches to manage virtual memory on a system: paging and segmentation. Paging−based systems divide virtual memory into equal−size pages, only some of which need to reside in main memory in order to execute a program. Segmentation−based systems use variable−length segments, only some of which need to reside in main memory. Some systems use a combination of both paging and segmentation, so that only some pages of some segments need to reside in main memory. Both paging and segmentation have been in use for almost 50 years [Fotheringham 1961; Burroughs 1961].

5.1 ix86 Memory segmentation

The ix86 architecture supports both memory paging and segmentation [Intel 2008]. Each memory address space consists of one or more memory segments, where a memory segment is defined by an entry in a local descriptor table (LDT) with a base address, segment length, and access permissions (see figure 6). Physical addresses are computed by using the appropriate segment register value as an index into the LDT and adding the corresponding base address to the virtual address (the ix86 has separate segment registers for code, data, stack, plus three other segments). While updating the LDT is a privileged operation, updating segment registers is not.

Early work on ix86 sandboxing by Chiueh et al. required kernel modifications [Chiueh 1999]. Modern versions of Linux, BSD, Mac OS X, and Microsoft Windows implement system calls that provide an unprivileged application with limited control over its LDT. By using memory segments as isolation domains, it becomes possible to safely execute untrusted code without the overhead of address checking instructions, and to switch between isolation domains without the cost of a hardware context switch.

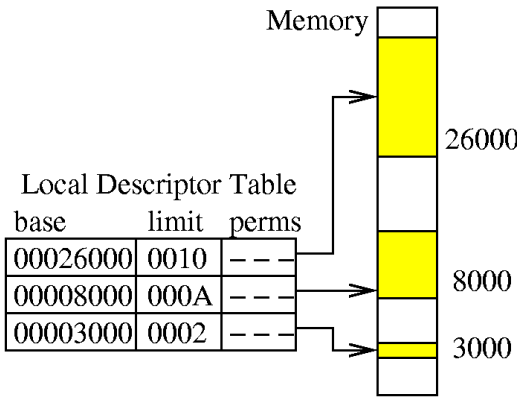

Figure 6. ix86 Local descriptor table and memory organization (after [Irvine 2007]).

The ix86 segmentation feature is available only in 32−bit mode; 64−bit host applications can still use this to run 32−bit code.

5.2 VX32 − User−level sandbox with hardware support

VX32 implements a sandbox that runs in the context of a host application [Ford 2008]. It uses a combination of ix86 segmentation and light−weight binary instrumentation to isolate a less trusted guest application from its host. VX32 supports multiple threads: each thread executes its own sandbox, and sandboxes may share data memory. This approach is independent of the application implementation language.

A light−weight on−the−fly code translator enforces confinement: guest applications are not allowed to manipulate segment registers or to make direct system calls. VX32 ensures that jumps, calls and returns go to the start of translated code blocks, which are kept in a separate segment. In guest applications, data access (including stack) happens at native speed, while indirect jumps (including function returns) and Linux system calls are 4−5x slower. The sandbox footprint is 3800 lines of C and 500 lines of assembly, half of which make up the code translator.

Besides isolating guest memory accesses from its host, VX32 virtualizes the guest's system call API including signals, and delegates system call execution and signal delivery to the host application. This can be used not only to confine application code, but also to implement application binary portability, i.e. to run the same application binary code on different host OS environments. VX32 host implementations exist for Linux, FreeBSD and MacOS X.

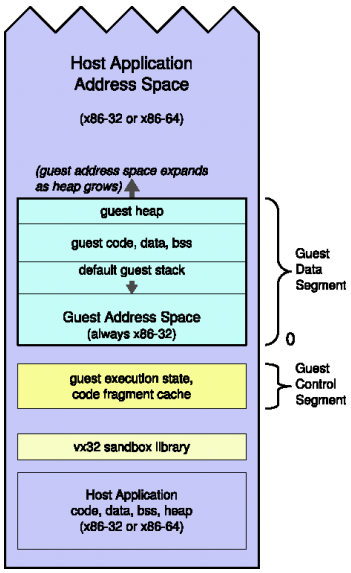

Figure 7. VX32 Memory organization (from [Ford 2008]).

Examples of application binary portability are VXA, an archival system that provides portable extraction utilities [Ford 2005]; Alpaca, an extensible PKI framework [Lesniewski−Laas 2007]; and Plan9 VX, a user−level port of the Plan 9 operating system [Pike 1995] that can run unmodified Plan 9 binaries on any system that has a VX32 implementation.

SPEC CPU2006 application benchmarks show a run−time overhead that varies with the CPU model; the worst case is less than 10% for tests with few indirect branches, and 50−70% otherwise. Due to improved code locality, some benchmarks run up to 20% faster. Macro performance tests with VXA decoders show runtime overheads from −30% to +30% on a variety of CPU models.

To protect the sandbox, VX32 does not allow guest applications to manipulate segmentation registers. Thus, VX32 is not compatible with applications that manipulate segment registers such as 16−bit applications, and with VX32 itself. VX32 is also not compatible with thread libraries that use segments to store thread−local data. It would have to be extended to allow guests to create new segments in a controlled manner.

Like many systems that translate code, VX32 is currently not compatible with some applications that generate code at run−time. It translates such code into a safe version for execution, but there is currently no mechanism to invalidate cached translations when the application changes the original.

5.3 Google Native Client

The Google native client (NaCl) implements an unprivileged sandbox for ix86 mobile web applications [Yee 2008]. Sandbox implementations exist for Linux, Windows and MacOS X host environments. NaCl relies on a combination of SFI for CISC (section 3.1.2) plus ix86 segmentation−based hardware isolation (section 5.1). Like

VX32, the sandbox provides an OS−independent system call interface and can run the same application binary on different host OS environments.

The implementation consists of an untrusted compiler toolchain that produces statically−linked executables, a trusted verifier that inspects guest binary code at load time, and trusted runtime support code that manages the sandbox and that provides an infrastructure for communication with web applications and other NaCl sandboxes. The amount of trusted code is about 25000 lines, of which the trusted verifier takes up about 500 statements. Guest binary code must adhere to a simple format, so that all "forbidden" instructions can be found in one monotic scan over the code memory segment. A second scan verifies that all direct jump instructions will transfer control to instructions that were found during the first scan.

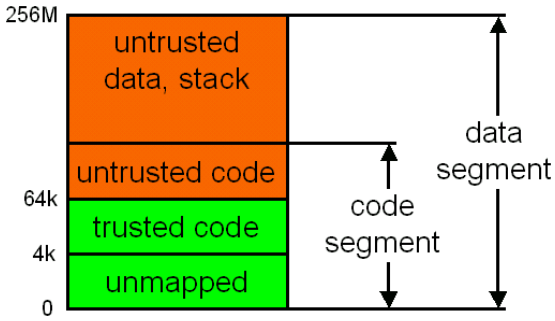

Figure 8. Memory layout for native−client guest applications. The untrusted regions contain guest application read−only code, read−write data and per−thread stacks. The trusted region contains read−only code to enter and return from system calls; the trusted read−write data and per−thread stacks are outside guest−accessible memory.

The NaCl implementation makes the following performance−oriented changes over the SFI−for−CISC implementation discussed in section 3.1.2. 1) To restrict the targets of indirect jumps, calls and returns, NaCl increases the instruction memory block size from 16 to 32 bytes, and forces indirect jump target addresses to be multiples of 32; the larger instruction memory block size reduces the amount of instruction padding. 2) The data and code sandboxes are implemented with ix86 segmentation−based hardware isolation. This eliminates the need for address "wrap around" instructions.

Thanks to this implementation, NaCl performance compares favorably with that of section 3.1.2. With SPEC2000 benchmarks, application code expansion is reduced from an average of 70% to less than 50%; the run−time overhead is reduced from an average of 21% for load/jump sandboxing to 5% on average for full sandboxing (i.e. load/store/jump). The NaCl implementation uses an external system−call sandbox (section 4) as a safety net, in case the segmentation−based sandbox fails.

6 Resource Virtualization

The preceding sections focused on interposition as a means to confine program

execution, by forbidding direct access to external resources. Here we discuss interposition as a means to virtualize a sandboxed program's view of its environment.

The simplest approach hides resources that must not be visible from inside a sandbox; from the sandboxed code's point of view, the hidden resources do not exist. A more sophisticated approach translates between virtual names inside the sandbox and physical names outside. We will discuss examples of both approaches.

Resource virtualization may be achieved at any suitable interface in the system architecture. Virtualization at the library interface is suitable only for cooperative applications, and won't be discussed in this report. Instead we will focus on virtualization techniques that can be enforced at the operating system and hardware interfaces.

6.1 Operating system−level virtualization

With virtualization at the operating system interface, multiple execution environments can exist side by side on top of the same operating system kernel. Each execution environment provides processes with its own view of the file system, network, and other resources. Execution environments can have their own super−users, although only the default execution environment can perform operations that affect global system properties such as kernel and network configuration.

As a general rule, the runtime cost of kernel−based implementations is only a few percent, while user−land implementations are an order of magnitude more expensive due to the additional control transfers to a user−land monitor process.

By avoiding the need for one operating system instance per execution environment, OS−level virtualization can achieve better scalability than hardware virtualization, especially when execution environments contain only a limited number of processes and files. For OS−level virtualization to be successful in production settings, the underlying system needs to be able to provide strong performance guarantees.

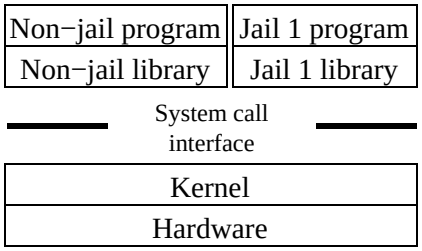

Figure 8. Operating system−level virtualization showing the default (non−jail) execution environment and one "jail" execution environment.

Without system call translation, OS−level virtualization is limited to applications that can run directly on the host operating system. For example, BSD or Solaris systems can run executables from a limited number of other operating systems including Linux. With system call translation, a wider range of application software can be supported.

6.1.1 UNIX chroot

The UNIX chroot (change root) system call is a privileged operation that changes the root directory of the calling process. Although chroot is often used to harden systems, security was not its initial purpose [Kamp 2002]. To enter a chroot tree, a privileged process invokes the chroot system call with the pathname of the new root directory, and changes its current directory to a location under the new root directory. chroot does not change the semantics of non−file system calls, such as calls that manipulate processes, networking, or kernel configuration. For this reason, chroot is not suitable to confine hostile privileged users. Noteworthy applications of chroot are the anonymous FTP service and the Postfix mail server.

6.1.2 FreeBSD 4.0 Jails

The FreeBSD jail system call [Kamp 2000] takes the chroot concept further, and provides support for multiple execution environments on the same kernel instance. Each jail hides processes and files outside the jail, and stamps network packets with the IP address that is assigned to the jail. To instantiate and enter a jail, a privileged process specifies a root directory, hostname, and IP address; a privileged process may also enter an existing jail by specifying its name. There is no "boot" procedure as with Solaris zones (next section). The name hiding mechanism affects only processes that execute inside jails, and does not hide resources inside jails from non−jailed processes. The code footprint is about 400 new lines of kernel code with negligible performance impact.

Jails confine privileged processes by disallowing system calls that change kernel configuration, network configuration, and that create device entry points in the file system. In practice, jails have a few limitations that are direct consequences of their design. Since jails are primarily based on name hiding instead of name translation, they don't prevent name clashes in the global System V IPC name space. Another limitation is that jails must share an IP address with the non−jail environment, in order to communicate with the network. Thus, network applications in the non−jail environment may accept or create connections on the IP address that is shared with a jail.

6.1.3 Solaris 10 zones

Solaris zones [Price 2004] are implemented with name hiding similar to FreeBSD jails. Unlike jails, Solaris zones contain an entire file system tree (with the exception of kernel files). Each running zone has a kernel−resident process (zsched) which holds zone−specific state, and a user−land process (init) which executes the zone's startup and shutdown procedures. Solaris comes with tools to configure and instantiate zones, and its package manager is made zone−aware so that package management works as expected.

As with FreeBSD jails, super−users in zones are not allowed to make system calls that would be harmful to other zones, and resources inside zones are visible from the default zone. Unlike FreeBSD jails, resources inside zones are not accessible to unprivileged users in the default zone, nor can processes in the default zone listen on

the IP address that is assigned to a zone. The initial Solaris zone implementation supports CPU resource control per zone, while later versions also support memory resource control.

According to [Price 2004], runtime overhead for applications inside is zones less than 1%; in practice, the performance is dominated by other factors such as the use of loop−back file system mounts to conserve disk space.

6.1.4 PeaPod − namespaces and sandboxes

PeaPod uses a combination of private name spaces (Pods) and system call sandboxes (Peas) to implement least privilege execution on Linux [Potter 2007]. The implementation has a typical runtime overhead of 4% for popular servers. In many respects, the PeaPod functionality is similar to that of Systrace plus a FreeBSD jail.

Pods (process domains) implement private name spaces by translating virtual names in system calls into physical names for the underlying host system, and vice versa. The translation involves a combination of chroot, system call interposition, and file system stacking. A pod name space is available only to processes that execute within that pod, and processes or files outside a pod are inaccessible from inside. For example, a web service can be implemented with a pod that contains only the necessary processes and files.

Peas (protection and encapsulation abstractions) enforce restrictions on system calls, but do not affect the visibility of virtual names inside the same pod. Peas label a pod's virtual identifiers with the identifier of the pea, by exploiting a combination of system call interposition and file system stacking. By default, processes within a pea can manipulate only processes within the same pea. Transition rules specify how a process can enter a different pea within the same pod, by executing a specific file. For example, the http daemon of the aforementioned web service can use this transition mechanism to run cgi programs under control of a different pea.

Peas take advantage of file system stacking to avoid file system race conditions. Stacking provides a way to hook into the file system's lookup and permission functions; this ensures that pea policies and system calls apply to the same file system objects. However, this does not eliminate memory race problems with non−file system calls that have pointer−valued arguments, such as pointers to network address data structures.

6.1.5 System call translation

Besides hiding or translating resource names, system call interposition can also be used to translate system call APIs from alien operating system environments into their local equivalents. System call translation has a long history. See [Jones 1993] for earlier examples of implementations that provide one operating system interface on top of a different one. What follows is a short list of contemporary open source implementations.

- Wine [Winehq 2008] implements the Windows API on top of UNIX−like

systems via API translation, and comes with its own versions of Windows DLL files, registry and other infrastructure.

- The VX32 sandbox [Ford 2008], introduced in Section 5.2, implements system call delegation with system call translation. Although its primary goal is to support portable applications that use only a few operating system services, VX32 has also been used to implement an execution environment for software that was built to run on the Plan 9 operating system.

- The Native Client (NaCl) sandbox [Yee 2008], introduced in section 5.3, implements system call delegation with system call translation. In this case the purpose is to run portable guest code that is part of a web application, where the web browser provides most of the infrastructure and policies to access remote sites and local resources.

6.1.6 Other OS−level virtualization approaches

Sysjail [Dzonsons 2006] implements FreeBSD jail−style execution environments on NetBSD and OpenBSD kernels; this is a user−land implementation that builds on the Systrace system call monitoring infrastructure.

Linux OpenVZ [OpenVZ 2008] and Linux−VServer [VServer 2008] implement multiple execution environments on Linux kernels; like FreeBSD jails and Solaris zones, these are kernel−based implementations. Noteworthy is that OpenVZ implements name translation for process identifiers.

FVM [Yu 2006] is a prototype that implements multiple execution environments on Windows; it uses copy−on−write name space translation and is implemented as a kernel−level system call wrapper plus user−level library wrapper.

This area is also covered by commercial browser virtualization products such as GreenBorder (acquired by Google), as well as products that implement application virtualization, also known as "application streaming".

6.2 Host−level virtualization

Virtual machine monitors (VMMs) provide virtual hardware environments for guest operating systems, with (mostly) the same CPU instruction set architecture as the underlying host's physical hardware. Host virtualization has recently become popular on commodity hardware with implementations such as VMware [Sugerman 2001], Xen [Barham 2003], User−mode Linux [Dike 2001], Denali [Whitaker 2002] and QEMU [Bellard 2005].

Conceptually, VMMs implement virtualization via resource name translation at the OS/hardware boundary. VMMs interpose on requests for memory, disk and other hardware resources, and translate between virtual names in the guest and physical names in the host. Isolation of guest resources is necessary, because operating systems do not expect to write−share their low−level resources such as disk blocks or memory pages with other operating system instances.

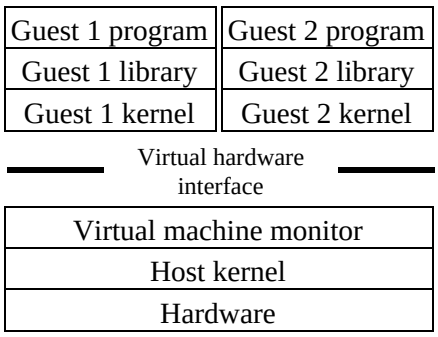

Figure 9. Typical software virtual machine architecture. Some implementations such as Xen [Barham 2003] run the virtual machine monitor on bare hardware with assistance of a privileged partition that contains a conventional operating system, and some implementations such as VMware workstation [Sugerman 2001] run as an application on top of a conventional host operating system.

6.2.1 VMM security

In practice, resource isolation is not only a matter of resource name translation. VMMs for commodity platforms are complex systems that run at the highest privilege level, and that unavoidably introduce vulnerabilities of their own. According to a recent study, current VMM implementations are particularly prone to have errors in their handling of malformed requests from guests to emulated hardware devices [Ormandy 2007].

VMM security can significantly suffer from dependencies on other software. Implementations such as Xen and VMware workstation target a wide range of hardware, and rely on a conventional operating system to provide the necessary device driver support. This increases the privileged code footprint with millions of lines of code. The security of these VMM's is therefore no stronger than the security of the conventional operating system that supports the VMM [Wojtczuk 2008, Karger, 2008].

6.2.2 VMM performance

VMMs maintain control over physical hardware by trapping interrupts and privileged instructions. Some VMM implementations require guest software modifications, an approach that is called paravirtualization. Other implementations support unmodified guest operating systems; this approach is called full virtualization. The two approaches differ in performance and in complexity.

Paravirtualization can work around non−virtualizable instructions (sensitive instructions that aren't trapped by the CPU) and can eliminate performance bottlenecks by replacing multiple traps by a single call into the VMM. Full virtualization requires dynamic binary translation of non−virtualizable instructions in kernel code; this instrumentation of kernel code is called on−the−fly paravirtualization.

Virtualization introduces significant overhead when it is implemented by trapping every individual privileged instruction. For this reason alone, binary translation and paravirtualization remain relevant solutions even after fully virtualizable CPUs have become available. Both Xen and VMware currently promote paravirtualization interfaces.

6.2.3 Video performance

One area that can benefit from paravirtualization is video output. Graphics processing units (GPUs) provide accelerated performance for demanding tasks such as three−dimensional rendering or streaming video, but their hardware is not always documented in public, and open−source drivers are not always available.

Lagar−Cavilla et al. [Lagar−Cavilla 2007] find that 86% or more of non−virtual performance can be achieved by virtualizing the high−level OpenGL library API [OpenGL 2008], instead of virtualizing low−level and poorly−standardized hardware interfaces. VMware Workstation takes a similar approach with DirectX [Microsoft DirectX 2008; VMware Direct3D 2008].

With Tahoma [Cox 2006], guest machines use a special version of the Qt graphics library to achieve 60% of local−application streaming video throughput. Other work on video performance in distributed environments has found improved display performance up to LAN speeds by hooking intermediate−level primitives [Baratto 2005].

The above approaches communicate requests over conventional network protocols. With everything running on the same physical hardware, VMware's SVGA guest driver takes advantage of shared memory and uses virtual DMA to speed up communication with the host's GPU [Dowty 2008].

6.2.4 The Tahoma web browser

The Tahoma web browser [Cox 2006] uses host virtualization to isolate different web applications that run on behalf of the same user. Each web domain runs in its own unprivileged Linux guest machine under the Xen VMM. The privileged "domain 0" guest machine runs a monitor that manages resources on behalf of web applications, and runs the browser's window manager. The code footprint is about 10,000 lines.

Besides security, attention is given to application responsiveness and video performance. Given the heavy isolation mechanisms used, this is not a trivial matter. Tahoma uses pre−forked virtual machines with stock browsers that can open a new URL in 1.06 seconds (the latency is 0.84 seconds with the same browser on native Linux). With a special version of the Qt graphics library, Tahoma achieves about 60% of the streaming video throughput compared to local X11 applications; without these modifications, the same application in a virtual machine achieves only 5% of the video throughput with generic X11 over TCP/IP.

7 Other architectures

Although this report focuses on isolation techniques that may be used with today's applications and operating systems, it is worthwhile to draw lessons from work on related problems in other environments. This section lists a number of approaches that the author encountered during the research for this report.

The subject of multiple protection domains per address space has received attention from opposing camps; some propose to abandon hardware isolation altogether because of the cost of cross−domain communication; others firmly believe that hardware isolation is the only thing that really works; and some see opportunities to improve cross−domain communication performance without throwing out the child with the bath−water.

- Singularity [Aiken 2006] implements an operating system with software isolated processes, where memory safety is enforced by a combination of language type safety and runtime checks. Singularity's processes have non−overlapping memory ranges. Processes exchange messages by passing the exclusive use of references to memory blocks in an exchange heap.

  The absence of hardware isolation places high demands on the correctness of the compiler and of its application runtime support, including the in−process memory garbage collector and the IPC mechanism. Singularity's performance comes at the price of having to use a type−safe language with reduced functionality. With conventional systems, unsafe implementation languages make software isolation much more expensive.

- Mondriaan [Witchel 2005] is a hardware and software design for fine−grained memory protection between multiple protection domains within a single linear address space. If these features were to become available on commodity systems, they would benefit the safety of both kernel−level and application−level extensions.

- The authors of Opal [Chase 1992] see the advent of 64−bit computing as an opportunity to abandon per−process virtual address spaces, which they consider an artifact of architectures with limited address ranges. Instead, they advocate the use of one global virtual address space with multiple protection domains for persistent and non−persistent information.

  The idea is to separate the naming (the addresses) from the protection of those addresses. By making memory address translations context independent, the authors argue they can simplify the hardware memory management implementation. If this idea were to be implemented in commodity hardware, it would help to reduce the cost of cross−domain communication.

Another related field of research is concerned with information flow policies. Sofar, mandatory policies have not made much impact on commodity systems. What follows are examples of mandatory policies based on contamination labels.

- SubOS [Ioannidis 2001] explicitly targets web client security. It implements isolation by labeling downloaded files according to their origin, and by contaminating browser helper programs upon access to those files. Once contaminated, the helper's access rights are determined by the contamination label instead of the invoking user's identity. SubOS was implemented by making changes to an OpenBSD kernel.

- Asbestos [VanDeBogart 2007] uses mandatory and discretionary labels to control which services a process can invoke, and which processes it can communicate with. The labels are used to enforce information flow policies, by contaminating event−driven threads with the labels of the resources that they have accessed in the past.

The ideas behind Asbestos are developed in the context of large−scale server applications that handle information on behalf of many different users. However, the basic problem is similar to the one that SubOS targets: enforcing isolation policies on software that manipulates information on behalf of different parties.

8 Discussion

As mentioned in the introduction, the purpose of this report is to review mechanisms that are compatible with conventional operating systems, and that may be used to execute web applications in isolation from each other and from the host platform. Instead of covert channels, the primary focus is on preventing untrusted web applications from corrupting other applications or the host platform itself. Table 1 summarizes the findings for commodity desktop platforms.

Sandboxing level	Maturity	Generality	Performance	Security
Host	medium	OS−independent	medium	high privilege, 10^5 lines
Language runtime	high	language−dependent	high with JIT	unprivileged, 10^5 lines
Operating system	high to medium	OS−dependent	high to medium	10^3 lines
In−process	low	language−independent	high to medium	unprivileged, 10^4 lines

Table 1. Summary of sandbox approaches. See text for discussion.

Host−level sandboxes use virtualized hardware to encapsulate an entire host, including operating system and applications. This is the approach taken by the Tahoma web browser [Cox 2006]. However, from a security point of view, using a host−level sandbox violates the principle of least privilege, as it encapsulates an unprivileged application with a complex virtual machine monitor that runs at the highest privilege level. With contemporary VMM implementations, the poor handling of malformed guest requests to virtual devices is a cause for concern [Ormandy 2007], as is the critical dependency of such VMMs on conventional operating

systems [Wojtczuk 2008; Karger 2008].

Language−runtime sandboxes encapsulate an application within a software virtual machine. This approach deserves serious consideration especially where major web browser components are already available in Java [SUN Java 2008] or in languages supported by Microsoft's CLR [Microsoft CLR 2008]. This approach avoids the need for high−privileged code, although the trusted code footprint can be of the order of 10^5 lines [Appel 2002]. The author has insufficient implementation experience in this area to judge the practicality of these languages for general browser infrastructure programming.

Operating system−level sandboxes encapsulate applications at the operating system kernel interface, and share same the kernel with sandboxed and non−sandboxed applications. This approach is less mature, but it introduces only a small amount of trusted code, and is therefore attractive from a security point of view. This is the approach taken by the OP browser [Grier 2008], Google Chromium [Chromium Sandbox 2008], and Ostia [Garfinkel 2004]. OS−level sandboxes can have code footprints as small as several thousand lines.

In−process sandboxes encapsulate an untrusted guest application within the address space of a trusted host application. This approach is the least mature. It requires no privileged code, and can deliver good performance especially when combined with hardware segmentation support [Ford 2008, Yee 2008]. Some implementations provide a generic OS−independent system call API that allows the same binary application to run on multiple host OS environments. The amount of trusted code can be as small as several thousand lines. However, in−process sandboxing can be incompatible with software that generates executable code on the fly, a trend that is becoming popular with, for example, high−performance JavaScript implementations. Both the Flash and Acrobat multimedia plugins contain JavaScript engines, in addition to the JavaScript engines that already reside in the web browsers themselves.

9 Conclusion

This report looked at mechanisms that can harden web browsers for contemporary operating systems, by partitioning browsers into subsystems that execute with reduced privileges. The sobering finding is that operating system security, in the form of OS−level sandboxes, provides a comparatively good return on investment.

Although virtual machine monitors can provide a high degree of isolation [Karger 1991], VMM implementations for desktop environments are far from small, they run with the highest privilege level, and they critically depend on conventional operating systems for device driver and management support. It makes little sense to use such VMMs to sandbox unprivileged application code.

Recent applications of legacy ix86 segmentation have led to a revival of techniques from mainframe days to run a binary application on top of an alien host operating system [Ford 2008, Yee 2008]. This is ongoing research, and it will take a few years before the technology is suitable for general use.

References

[Abadi 2005a] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control−Flow Integrity: Principles, Implementations, and Applications. ACM Conference on Computer and Communication Security (CCS), Alexandria, VA, USA, November 2005.

[Abadi 2005b] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. A Theory of Secure Control−Flow. International Conference on Formal Engineering Methods (ICFEM), Manchester, UK, November 2005.

[Acharya 2000] Anurag Acharya and Mandar Raje. MAPbox: Using Parameterized Behavior Classes to Confine Applications. 9th USENIX Security Symposium, Denver, CO, USA, August 2000.

[Aiken 2006] Mark Aiken, Manuel Faehndrich, Chris Hawblitzel, Galen Hunt, and James Larus. Deconstructing Process Isolation. 2006 Workshop on Memory system performance and correctness, San Jose, CA, USA, 2006.

[Appel 2002] Andrew W. Appel and Daniel C. Wang. JVM TCB: Measurements of the Trusted Computing Base of Java Virtual Machines. Technical Report Technical Report CS−TR−647−02, Princeton University, 2002.

[Barham 2003] 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. 19th ACM symposium on Operating systems principles, pp 164−177, Bolton Landing, NY, USA, 2003.

[Barth 2008] Adam Barth, Collin Jackson, Charles Reis, and the Google Chrome Team. The Security Architecture of the Chromium Browser. Technical report. http://crypto.stanford.edu/websec/chromium/

[Bellard 2005] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. USENIX 2005 Annual Technical Conference, FREENIX Track, pp. 41−46, Anaheim, CA, USA, 2005.

[Berman 1999] Andrew Berman, Virgil Bourassa and Erik Selberg. TRON: Process−Specific File Protection for the UNIX Operating System. USENIX 1995 Winter Technical Conference, New Orleans, LA, USA, January 1995.

[Burroughs 1961] Burroughs Corp. The Descriptor −− a definition of the B5000 Information Process ing System. Bulletin 5000−20002−P, Detroit, 1961. http://www.cs.virginia.edu/brochure/images/manuals/b5000/descrip/descrip.html

[Chari 2002] Suresh N. Chari and Pau−Chen Cheng. BlueBoX: A Policy−driven, Host−Based Intrusion Detection system. Network and Distributed Systems Security Symposium (NDSS), San Diego, CA, USA, February 2002.

[Chen 2005] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non−Control−Data Attacks Are Realistic Threats. In Proceedings of the 14th USENIX Security Symposium, pages 177−192. Baltimore, MD, USA, 2005.

[Chiueh 1999] Tzi−cker Chiueh, Ganesh Venkitachalam, and Prashant Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. 17th ACM Symposium on Operating Systems Principles (SOSP), Kiawah Island, SC, USA, December 1999.

[Chou 2001] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An Empirical Study of Operating Systems Errors. 18th ACM Symposium on Operating Systems Principles (SOSP), October 2001.

[Chromium Sandbox 2008] Chromium Developer Documentation: Sandbox. http://dev.chromium.org/developers/design−documents/sandbox. Last visited November 2008.

[De Keukelaere 2007] Frederik De Keukelaere, Sumeer Bhola, Michael Steiner, Suresh Chari, and Sachiko Yoshihama. SMash: Secure cross−domain mashups on unmodified browsers. 17th International World Wide Web Conference (WWW), 2008.

[Dike 2001] Dike, Jeff. User−mode Linux. 5th Annual Linux Showcase & Conference (ALS 2001), Oakland, CA, USA, 2001.

[Dzonsons 2006] Kristaps Dzonsons. Sysjail: systrace userland virtualisation. NYBSDCON conference, New York, NY, USA, October 2006.

[Erlingsson 2006] Ulfar Erlingsson, Martin Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software Guards for System Address Spaces. 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI),

Seattle, WA, USA, November 2006.

[Erlingsson 2007] Ulfar Erlingsson. Methods and systems for providing a secure application environment using derived user accounts. United States Patent #7191469, March 13, 2007.

[Ford 2005] Bryan Ford. VXA: A virtual architecture for durable compressed archives. 4th USENIX FAST, San Francisco, CA, USA, December 2005.

[Ford 2008]. Bryan Ford and Russ Cox. Vx32: Lightweight User−Level Sandboxing on the x86. USENIX 2008 Annual Technical Conference, Boston, MA, USA, June 2008.

[Fotheringham 1961] Fotheringham, J. Dynamic storage allocation in the Atlas computer, including an automatic use of a backing store. Comm. ACM 4, 10 (Oct. 1961), 435−436. http://www.eecs.harvard.edu/cs261/papers/frother61.pdf

[Fraser 1999] Timothy Fraser, Lee Badger, and Mark Feldman. Hardening COTS Software with Generic Software Wrappers. 1999 IEEE Symposium on Security and Privacy, Oakland, CA, USA, May 1999.

[Garfinkel 2003] Tal Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. Network and Distributed Systems Security Symposium (NDSS), San Diego, CA, USA, February 2003.

[Garfinkel 2004] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A Delegating Architecture for Secure System Call Interposition. Network and Distributed Systems Security Symposium (NDSS), San Diego, CA, USA, February 2004.

[Goldberg 1996] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A Secure Environment for Untrusted Helper Applications. 6th Usenix Security Symposium, San Jose, CA, USA, July 1996.

[InformationWeek 2008] Google Chrome Answers The GreenBorder Mystery. InformationWeek, September 1, 2008.

[Intel 2008] Intel(R) 64 and IA−32 Architectures Software Developer's Manual, September 2008.

[Ioannidis 2001] Sotiris Ioannidis and Steven M. Bellovin. Building a Secure Web Browser. FREENIX track of the 2001 USENIX Annual Technical Conference, Boston, MA, USA, June 2001.

[Irvine 2007] Kip R. Irvibe. Assembly Language for Intel−based Computers. 5th ed. Prentice−Hall, 2007.

[Jaing 2000] K. Jain and R. Sekar. User−Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement. Network and Distributed Systems Security Symposium (NDSS), San Diego, CA, USA, February 2000.

[Jones 1993] Michael B. Jones. Interposition Agents: Transparently Interposing User Code at the System Interface. 14th ACM Symposium on Operating Systems Principles (SOSP), Asheville, NC, USA, December 1993.

[Kamp 2000] Poul−Henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. 2nd International SANE Conference, Maastricht, The Netherlands, May 2000.

[Kamp 2002] Poul−Henning Kamp. Rethinking/dev and devices in the UNIX kernel. BSDCon '02 Conference on File and Storage Technologies, San Francisco, CA, USA, February 2002.

[Karger 1991] Paul A. Karger, Mary Ellen Zurko, Douglas W. Bonin, Andrew H. Mason, and Clifford E. Kahn. A Retrospective on the VAX VMM Security Kernel. EEE Transactions on Software Engineering archive Volume 17, Issue 11 (November 1991), Pages: 1147 − 1165.

[Karger 2008] Paul A. Karger. Is Your Virtual Machine Monitor Secure? Keynote presentation at the International Forum on Trusted Infrastructure Technologies and 3rd Asia−Pacific Trusted Infrastructure Technologies Conference (APTC 2008).

[Kennedy 2008] Nial Kennedy. The story behind Google Chrome. Blog posting at http://www.niallkennedy.com/blog/2008/09/google−chrome.html, September 2, 2008.

[Lagar−Cavilla 2007] H. Andres Lagar−Cavilla, Niraj Tolia, M. Satyanarayanan and Eyal de Lara. 3rd International ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE), San Diego, CA, USA, June 2007.

[Lesniewski−Laas 2007] Chris Lesniewski−Laas, Bryan Ford, Jacob Strauss, Robert Morris, M. Frans Kaashoek, Alpaca: extensible authorization for distributed services, 14th ACM conference on Computer and communications security (CCS), Alexandria, VA, USA, October2007.

[Liang 2003] Zhenkai Liang, V.N. Venkatakrishnan and R. Sekar. Isolated Program Execution: An Application Transparent Approach for Executing Untrusted Programs. 19th Annual Computer Security Applications Conference (ACSAC 2003), Las Vegas, NV, USA, December 2003.

[McCamant and Morrisett 2006] Stephen McCamant, Greg Morrisett. Evaluating SFI for a CISC Architecture. 15th USENIX Security Symposium, Vancouver, BC, Canada, August 2006.

[Microsoft CLR 2008] Common Language Runtime Overview. Microsoft Developer Network (MSDN) website at http://mdsn.microsoft.com/. Last visited November 2008.

[Microsoft DirectX 2008] Microsoft DirectX website, http://msdn.microsoft.com/en−us/directx/default.aspx. Last visited November 2008.

[Mozilla 2008] The Mozilla foundation, Security Advisories for Firefox 2.0. http://www.mozilla.org/security/known−vulnerabilities/firefox20.html Last visited November 2008.

[OpenGL 2008] OpenGL website, http://www.opengl.org/. Last visited November 2008.

[OpenVZ 2008] The OpenVZ website. http://openvz.org/. Last visited ed November 2008.

[Ormandy 2007] Tavis Ormandy. An Empirical Study into the Security Exposure to Hosts of Hostile Virtualized Environments. CanSecWest Applied Security Conference, Vancouver, BC, Canada, April 2007.

[Peterson 2002] David S. Peterson, Matt Bishop, and Raju Pandey. A Flexible Containment Mechanism for Executing Untrusted Code. 11th USENIX Security Symposium, San Francisco, CA, USA, August 2002.

[Pike 1995] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. Computing Systems, Vol. 8, 3, pp. 221−254, Summer 1995.

[Potter 2007] Secure Isolation of Untrusted Legacy Applications. In Proceedings of the 21st Large Installation System Administration Conference, Dallas, TX, USA, November 2007.

[Price 2004] Daniel Price and Andrew Tucker. Solaris Zones: Operating System Support for Consolidating Commercial Workloads. 18th Large Installation Systems Administration Conference, Atlanta, GA, USA, November 2004.

[Provos 2003] Niels Provos. Improving Host Security with System Call Policies. 12th USENIX Security Symposium, Washington, DC, USA, August 2003.

[Provos 2008] Niels Provos, Panayiotis Mavrommatis, Moheeb Abu Rajab, and Fabian Monrose. All your iFRAMEs Point to Us. 17th USENIX Security Symposium, San Jose, CA, USA, August 2008.

[Reis 2006] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. BrowserShield: Vulnerability−Driven Filtering of Dynamic HTML. 2006 USENIX Symposium on Operating Systems Design and Implementation (OSDI), Seattle, WA, USA, November 2006.

[SUN Java 2008] The SUN Java website at http://java.sun.com/. Last visited November 2008.

[Singh 2008] Kapil Singh and Wenke Lee. On the Design of a Web Browser: Lessons learned from Operating Systems. Web 2.0 Security & Privacy 2008 (W2SP 2008), Oakland, CA, USA, May 2008.

[Small and Seltzer 1996] Christopher Small and Margo Seltzer. A Comparison of OS Extension Technologies. USENIX 1996 Annual Technical Conference, San Diego, CA, USA, January 1996.

[Small and Seltzer 1998] Christopher Small and Margo Seltzer. MiSFIT: A tool for constructing safe extensible C++ systems. IEEE Concurrency: Parallel, Distributed and Mobile Computing, 6(3), 1998.

[Sugerman 2001] Jeremy Sugerman, Ganesh Venkitachalam, and Beng−Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. 2001 USENIX Annual Technical Conference, Boston, MA, USA, June 2001.

[Switch 2003] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. 19th ACM Symposium on Operating Systems Principles (SOSP), Bolton Landing, NY, USA, October 2003.

[VServer 2008] The Linux VServer website. http://linux−vserver.org/, last visited November 2008.

[VanDeBogart 2007] Steve VanDeBogart, Petros Efstathopoulos, Eddie Kohler, Maxwell Krohn, Cliff Frey, David Ziegler, Frans Kaashoek, Robert Morris, and David Mazieres. Labels and Event Processes in the Asbestos Operating System. ACM Transactions on Computer Systems (TOCS) Vol. 25, No. 4, December 2007.

[VMware Direct3D 2008] VMware, Inc. Experimental Support for Direct3D. http://www.vmware.com/support/ws5/doc/ws_vidsound_d3d.html. Last visited December 2008.

[Wahbe 1993] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software−based fault isolation. 14th ACM Symposium on Operating Systems Principles (SOSP), Asheville, NC, USA, December 1993.

[Wang 2007] Helen J. Wang, Xiaofeng Fan, Collin Jackson, and Jon Howell. Protection and Communication Abstractions for Web Browsers in MashupOS. 21st ACM Symposium on Operating Systems Principles (SOSP), Stevenson, Washington, USA, October 2007.

[Watson 2007] Robert N. M. Watson. Exploiting Concurrency Vulnerabilities in System Call Wrappers. First USENIX Workshop on Offensive Technologies, Boston, MA, August 2007.

[Whitaker 2002] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and Performance in the Denali Isolation Kernel. 5th Symposium on Operating Systems Design and Implementation (OSDI), Boston, MA, USA, December 2002.

[Winehq 2008] The Winehq website at http://www/winehq.org/. Last visited November 2008.

[Witchel 2005] Emmett Witchel, Junghwan Rhee, and Krste Asanovic. Mondrix: Memory Isolation for Linux using Mondriaan Memory Protection. 20th ACM Symposium on Operating Systems Principles (SOSP), Brighton, UK, October 2005.

[Wojtczuk 2008] Rafal Wojtczuk. Subverting the Xen Hypervisor. Black Hat Briefings, Las Vegas, NV, USA, August 2008.

[Yee 2008] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. http://nativeclient.googlecode.com/svn/trunk/nacl/googleclient/native_client/documentation/nacl_paper.pdf. Last visited December, 2008.