Vx32: Lightweight User-Level Sandboxing on the x86

Bryan Ford and Russ Cox
MIT CSAIL

Presented by:
Ashwin Chaugule
26th Sept 2008
Area

• Systems security: Application Virtualization
• USENIX ’08 - Best Student Paper
• Source code: http://pdos.csail.mit.edu/~baford/vm/
  • (OR just google for vx32)
Problem

• Many applications are untrustworthy
• Inter application communication and with the kernel
• How do you mediate their operation?
Confinement

- **Sandboxing**: Host executes guest code in a confined environment, prevents it from affecting other code (app or host). Operations allowed are defined by policies.

- *principle of least privilege*

- Useful for security purposes as well as analysis
Related Work

• Capability Systems, L3’s Clan/Chief, Nested Processes etc
  • Problem: Expensive domain transitions

• Kernel space additions: Domain specific languages, type-safe languages, PCC etc
  • Problem: Requires a lot of rework
Related Work

• Similar work by Tzi-Chieuh, Ganesh et al
  • **Problem**: Modified kernel
• Sys-call Interpositioning: Janus etc
  • **Problem**: Requires sys-call API conformance with host OS
• TOCTOU problem, but that's solved by delegation based interpositioning
Problems

- Required too much rework
- Couldn’t handle variable length instructions
- Restricted by specific programming languages (type-safety)
- Had large overheads due to domain transitions
Vx32 VM

- Separate **code** sandboxing from **data** sandboxing

- **Data Sandboxing**: Through Segmentation hardware

- **Code Sandboxing**: Dynamic instruction translation (restrict code flow and can restrict ISA)
Memory Map

- ‘vxrun’ - contains elf-loader and inits space for guest
- vxrun hosts the guest

Figure 1: Guest and Host Address Space Structure
Data Sandboxing

- 6 segment registers available (no special privileges required)
- Segmentation cannot be disabled (just uses flat model: 0 - 2^{32} - 1)
Data Sandboxing

- 2 new segments in LDT per guest
  - Guest data segment
  - Guest control segment
- $ds, es, ss$ contain selectors to guest data segment
- with seg base addr 0, segmentation h/w auto confines data accesses within segment
Data Sandboxing

- Control Segment
- Hash table maps guest virt addr to code in fragment cache
- Code frag cache contains guest code
- fs or ss points to this segment

Figure 2: Guest Control Segment Structure
Code Sandboxing

- Main aim is to prevent guest from jumping out of the sandbox
- Segmentation only ensures data reads and writes are confined
- Need to trap segment overrides since those regs are unprivileged
Code Sandboxing

- **Key point**: Never execute guest code directly
- Transform that code into safe sequence and execute the sequence outside the box (code frag cache)
- ‘cs’ value always points to host app ‘vxrun’
  - doesn't change
Translation techniques

• Scan
  • scan guest code from ‘eip’ till unconditional branch or fragment size
  • gather extra info per instr: len, offset, type, worst-case translated size (constitute hint table)
Translation Techniques

- **Simplify**
  - Maintain control by restricting code flow to code frag cache
  - Scan hint table: direct jmps converted to intra-fragment jmps < if destination fragment exists in cache>
Translation Techniques

- **Place**: Compute offsets for patching direct jmps
- **Emit**: Actual patching or instruction fixups
  - Most instr’s are just copied
  - ‘unsafe’ ones are translated
  - branches and segment overrides
Translation Techniques

• All control transfer instructions patched to confine flow within safe sequence

• **Trampoline**: direct branches patched so that they jmp to hash lookup function

• Indirect branches can't be patched: Invokes lookup in hash table = major overhead
(a) An indirect jump to the address stored at 0x08049248:

```
08048160  jmp  [0x08049248]
↓
b7d8d0f9  mov  ebx, fs:[0x2c]
b7d8d100  mov  fs:[0x2c], ebx
b7d8d107  mov  ebx, [0x08049248]
b7d8d10d  jmp  vxrun_lookup_indirect
```

(f) A software interrupt:

```
08048160  int  0x30
↓
b7d8d0f9  mov  ebx, fs:[0x2c]
b7d8d100  mov  fs:[0x20], eax
b7d8d106  mov  eax, 0x230
b7d8d10b  mov  fs:[0x40], 0x8048162
b7d8d116  jmp  vxrun_gentrap
```

(b) A direct jump to 0x08048080:

```
08048160  jmp  0x08048080
↓
b7d8d0f9  mov  ebx, fs:[0x2c]
b7d8d100  jmp  0xb7d8d105
b7d8d105  mov  fs:[0x5c], 0x00008115
b7d8d110  jmp  vxrun_lookup_backpatch
b7d8d115  dword  0x08048080
b7d8d119  dword  0xb7d8d105
```
Exception Handling

• ‘eip’ now points to translated code, need to trace it original guest code for feedback

• Fragments already sorted in reverse order

• 1st Bin search: to get appropriate frag with ‘eip’

• 2nd Bin search: within frag’s hint table to get exact guest addr corresponding to ‘eip’

• get ‘eip’ with guest regdump
Evaluation
Figure 6: Normalized run times for SPEC CPU2006 benchmarks running under vx32. Each bar plots run time using vx32 divided by run time for the same benchmark running natively (smaller bars mark faster vx32 runs). The left three benchmarks use fewer indirect branches than the right four, resulting in less vx32 overhead. The results are discussed further in Section 4.3.
Evaluation
Applications

- **VXA**
  - exec decoders into compressed archive
  - vx32 protects host from malformed/buggy archives

- **Alpaca**
  - extensible PKI based on PCA
  - runs algos in sandbox

- **9vx**
  - Plan 9 OS ported
  - uses vx32 to launch userspace apps

- **Vxlinux**
  - delegation based interpositioning
  - relay guest syscalls to host OS
Figure 9: Normalized run times for VXA decoders running under vx32. Each bar plots run time using vx32 divided by run time for the same benchmark running natively (smaller bars mark faster vx32 runs). Section 5.1 gives more details. The jpeg test runs faster because the vx32 translation has better cache locality than the original code.
Figure 10: Normalized run times for cryptographic hash functions running under vx32. Each bar plots run time using vx32 divided by run time for the same benchmark running natively (smaller bars mark faster runs).
**Evaluation**

![Bar chart](image)

**Figure 11:** Normalized run times for simple Plan 9 benchmarks. The four bars correspond to Plan 9 running natively, Plan 9 VX, Plan 9 under VMware Workstation 6.0.2 on Linux, and Plan 9 under QEMU on Linux using the kqemu kernel extension. Each bar plots run time divided by the native Plan 9 run time (smaller bars mark faster runs). The tests are: swtch, a system call that reschedules the current process, causing a context switch (s1ee(0)); pipe-byte, two processes sending a single byte back and forth over a pair of pipes; pipe-bulk, two processes (one sender, one receiver) transferring bulk data over a pipe; rdwr, a single process copying from /dev/zero to /dev/null; sha1zero, a single process reading /dev/zero and computing its SHA1 hash; du, a single process traversing the file system; and mk, building a Plan 9 kernel. See Section 5.3 for performance explanations.
Take-Away

- Sandboxing is implemented without depending on kernel
- Hardware segmentation provides automatic sandboxing
- Dynamic translation overheads are mainly from indirect branches
- Vx32 satisfies: Complete mediation and Verifiability properties, but is it tamperproof? Run all userpace in vx32? init = vx32?
- Can we confine kernel extensions with this technique?
  - think user-level device drivers