StackGhost
Hardware Facilitated Stack Protection

Mike Frantzen – frantzen@(cerias.purdue.edu—nfr.com—openbsd.org)

Mike Shuey – shuey@ecn.purdue.edu

`http://StackGhost.cerias.purdue.edu`

CERIAS - Purdue University

NFR Security

OpenBSD

StackGhost at a Glance

- StackGhost can hinder exploits by protecting an application’s saved return pointer on the stack
- StackGhost is automatic and transparent to **ALL** processes on the system
- StackGhost imposes less than a **1%** performance penalty
- StackGhost is **not** another non-executable stack

Presentation Organization

1. Conventional function calls
2. Sparc function calls
3. StackGhost implementation
4. Performance
5. Limitations
6. Conclusion

Conventional Function Calls

1. Save registers before a function call
2. Save return pointer before a function call
3. Perform function CALL (or SYSCALL)
4. Restore saved registers after return

Saves and Restores result in slow memory accesses

Sparc function calls

Functions save the registers of its caller and restore them before returning via instructions:

- *SAVE* all of the registers
- *RESTORE* all of the registers

Sparc’s little kicker

- Defers using the stack in memory or cache
- Allocates a fresh set of private registers for each function call
- Leave previous functions’ registers intact
- To return, re-activate the last set of registers

Each set of registers is called a **“window”**

Register Windows

- The processor obviously does not have a limitless number of “register windows” available to allocate from. (Actually 6 or 7)
- In a deeply nested calling sequence, all the registers will be exhausted and some must be reclaimed.

Register Reclamation

1. The processor initiates a trap into the kernel.
2. The kernel saves the oldest register window into the userland stack.

The **kernel** writes **userland** registers (including the return pointer) to the stack.

Register Retrieval

When *RESTORE*ing registers, the processor will have to retrieve the window if the window was stored to the stack.

1. The processor initiates a trap into the kernel.
2. The kernel retrieves the register window out of the userland stack into the registers.

The **kernel** loads **userland** registers (including the return pointer) from the stack.

StackGhost

- An addition to the kernel register window *SAVE* and *RESTORE* trap handlers
- Automatically operates on return pointers before they are written to the stack
- Automatically operates on return pointers before they are popped off the stack

StackGhost Protection Methods

XOR Cookie

1. XOR a cookie into the process return pointers before they are stored to the stack.
2. XOR a cookie out of the process return pointers before they are loaded from the stack.

XOR Cookie Effects

How a XOR cookie inhibits exploits:

- Attacker cannot predict how the XOR cookie will affect the corrupted return pointer
- Steal the unused bits in the stored return pointer to carry a canary
- If the canary was corrupted, the window retrieval can abort

Per-Kernel XOR Cookie

First StackGhost incarnation:

- 13-bit sign extended XOR cookie.
- Includes 2 bits of canary.
- Cookie constant across every process.
- Costs 2 instruction per function call.

Will **not** stop a fully caffeinated attacker.

Per-Process XOR Cookie

Next Incarnation:

- 32-bit XOR cookie including 2 canary bits
- Cookie randomly generated per process.
- Cookie saved in 32-bit member of PCB.
- Costs 8 instruction per function call.

Will cause an exploit to branch to a random address; thus not work.

Return-Address Stack

The future:

- Keep return addresses in a stack inaccessible to processes.
- Place a unique random number where the return address normally goes in the user stack
- Stash the random number with the real return address

Return-Address Stack

During register window retrieval:

- Abort the process if the random number on the stack does not match the private copy
- Restore the return address from the private stack
- Restore the rest of the window like normal.

**** This has not yet been implemented ****

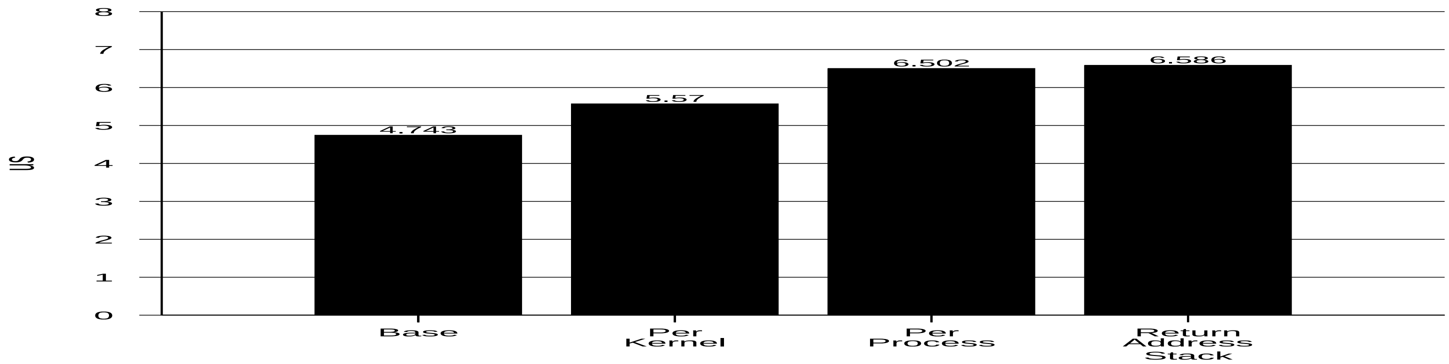
Method comparison

Chances that StackGhost will not explicitly detect a corrupt return pointer:

- Return-Address Stack: 1 in 2^{32}.
- XOR Cookie: 1 in 3

An exploit still may be foiled without explicit detection.

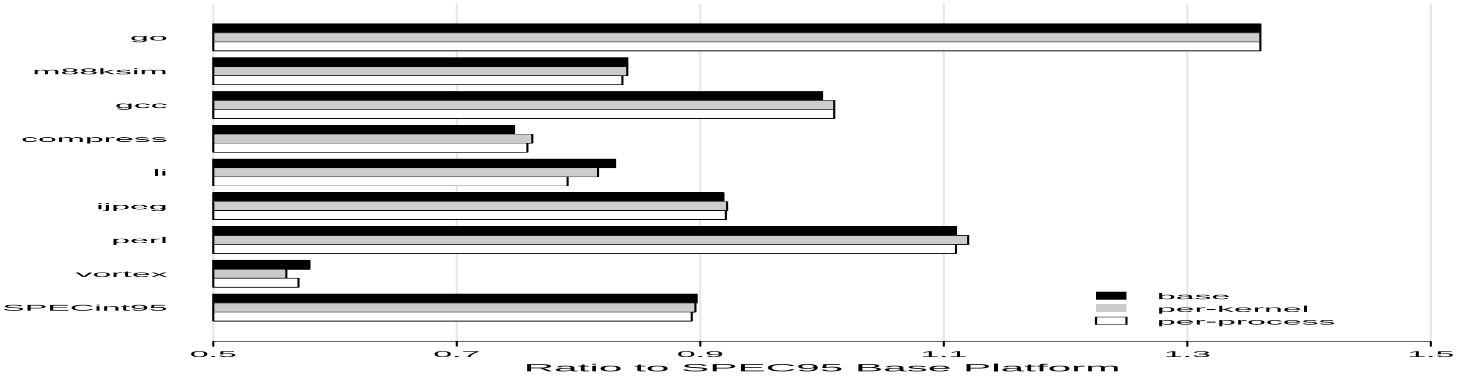
Micro Benchmarks



Absolute worst case overhead:

Per-Kernel Cookie	17.44%	
Per-Process Cookie	37.09%	
Return-Address Stack	38.86%	(approx)

SPECint95 Benchmarks



	% Overhead	SpecRatio
Stock OpenBSD	—-	0.897
Per-Kernel	0.1%	0.896
Per-Process	0.4%	0.893

StackGhost Limitations

- XOR Cookie’s cause unpredictable execution.
- Rootshell vs. DoS.
- Forked processes have identical Per-Process XOR Cookies. A new cookie is created during an execve().
- Debuggers are currently broken.
- No protection granularity. Protects all processes.

Conclusion

- StackGhost can transparently inhibit conventional attacks that overwrite the saved return address
- StackGhost cannot inhibit attacks that modify data, overwrite a function pointer etc.
- StackGhost is **NOT** a panacea. Correcting the bugs is better than depending on a crutch.

StackGhost
Hardware Facilitated Stack Protection

Mike Frantzen – frantzen@(cerias.purdue.edu—nfr.com—openbsd.org)

Mike Shuey – shuey@ecn.purdue.edu

`http://StackGhost.cerias.purdue.edu`

CERIAS - Purdue University

NFR Security

OpenBSD