StackGhost: Hardware Facilitated Stack Protection

Mike Frantzen
CERIAS
frantzen@cerias.purdue.edu

Mike Shuey
Engineering Computer Network
shuey@ecn.purdue.edu

Abstract

Conventional security exploits have relied on overwriting the saved return pointer on the stack to hijack the path of execution. Under Sun Microsystem's Sparc processor architecture, we were able to implement a kernel modification to transparently and automatically guard applications' return pointers.

Our implementation called StackGhost under OpenBSD 2.8 acts as a ghost in the machine. StackGhost advances exploit prevention in that it protects every application run on the system without their knowledge nor does it require their source or binary modification.

We will document several of the methods devised to preserve the sanctity of the system and will explore the performance ramifications of StackGhost.

1 Introduction

This paper presents a simple but elegant solution to the now infamous buffer overflow and some primitive format string attacks. Most security exploits have traditionally overwritten a function's saved return address. The attacker can then direct the flow of execution into an arbitrary instruction stream that is invoked when the vulnerable function tries to return control to its caller.

By taking advantages of one of the nuances of Sun Microsystem's Sparc processor architecture, we were able to engineer a kernel modification to OpenBSD 2.8 to help safeguard the return address. The kernel modification performs transparent, automatic and atomic operations on the return address before it is written to the stack and before the function transfers execution back to the saved return address.

Knowledge of what buffer overflows are [12], their relevance to security exploits [1, 13] and why they occur is a prerequisite to understanding this paper.

Section 2 describes the architectural issues involved in StackGhost. Section 3 details the implementation. Section 4 describes the performance effects. Section 5 acknowledges the limitations. Section 6 hypothesizes on extesion to other architectures. Section 7 describes the related research. Finally, Section 8 presents our conclusions.

2 Architectural Issues

2.1 Conventional Function Calls

Four bulk operations are performed to call a function in a conventional architecture. The function's parameters are saved onto the stack. The caller's registers are also saved onto the stack to prevent corruption by the callee. The instruction address is saved for the called function to return back to once it is finished. And only then can execution be transfered to the function.

Once the function completes its task, it jumps back to the return address saved on the stack.

2.2 Sparc Function Calls

2.2.1 Register Windows

When the Sparc architecture was first designed, the overhead associated with saving registers to the stack during a conventional function call was believed to be very large, or at least significant enough to warrant architectural changes to speed this process. Rather than wasting valuable CPU cycles to copy register data to and from the stack, Sparc architects attempted to provide hardware mechanisms to ensure that a function call gets a private set of registers for the duration of the function. When the function completes, the previous set of registers return to existence with (in most cases) no interaction with the stack whatsoever.

During normal execution, a Sparc processor has 32 visible general-purpose integer registers. These registers are divided into four groups based on the sort of data they are to contain, according to the Sparc Application Binary Interface (ABI) [17]:

global registers for data common across function calls.

input registers for incoming function parameters (including the frame pointer and return pointer).

local registers for general use.

output registers for parameters to deeper called functions, the return value from deeper function calls, the stack pointer, and the saved program counter after a jump and link.

The latter three groups (input, locals, and output) comprise a register window.

When a function is called, it allocates a new window for its specific use. The global registers are shared between both the old and the new windows (meaning that any modification of global data in the callee will be visible in the caller). The callee receives a new group of local registers, as well as a new set of output registers - these registers are not accessible from the calling function. Finally, the caller's output registers are rotated to be the input registers for the called function. Any changes the callee should make to its input registers will be visible to the caller as changes in the caller's output group of registers.

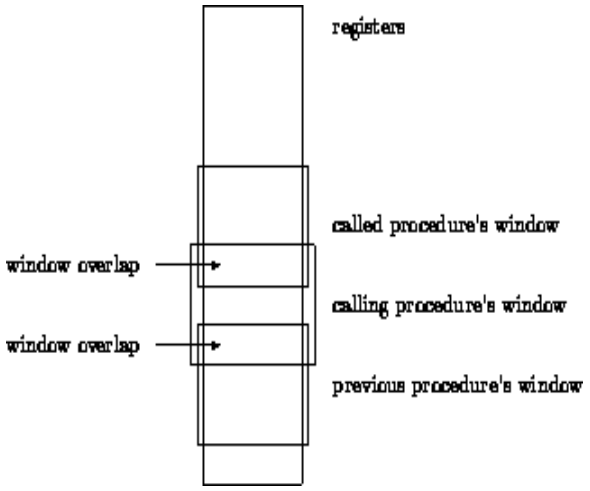

Figure 1: Register Window Overlap

In this way parameters can be passed from one function to another without (usually) interaction with the stack. The caller's code need only put parameters in its output registers, then call a function. The called function will have access to the caller's output registers in its own input registers. Return values are the reverse of this process; the called function leaves the return value in a particular input register, which then reverts to being an output register for the caller as soon as the function returns.

Nested function calls will create a chain of linked register windows. Each function call will use the same group of eight global registers, but will have its own group of eight local registers for its own private use. The output registers from the first function will be the input registers for the second deeper function called; the outputs from the second will be the inputs for the third, and so on.

Obviously, this trend can't go on forever. Each register window involves 24 registers (8 input, 8 local, 8 output), a third of which are shared with the calling function and two thirds of which need to be allocated by the processor. (The global registers are not shifted.) The processor will only have a limited number of registers available - most modern Sparc processors provide enough for seven or eight windows - and eventually some registers must be reclaimed.

The job of reclaiming registers falls to the operating system. When the number of allowable windows is about to be exceeded (as will occur with

any program exhibiting deeply-nested or recursive functions) a register window overflow interrupt is generated. The OS will respond by copying the oldest register window onto the stack, relocate the now defunct register window, and return control of execution to the program without it knowing it missed a beat. Eventually the deeply-nested functions in the program will start to complete but the caller's registers will be defunct and need to be fetched. The processor will generate a register window underflow interrupt and force the OS to restore the previously saved registers.

This OS interaction provides the basic hardware primitives needed for StackGhost's operation. In a conventional function call architecture, there is no feasible way for the OS to automatically examine critical areas of the stack as they are being written. However, because the OS is ultimately in charge of when registers are written to the stack on the Sparc architecture, it is possible to take extreme precautions to ensure the security of critical data, such as the return address and frame pointer.

2.2.2 Memory Alignment

StackGhost also takes advantage of one other Sparc architectural feature. Instructions must be aligned on a four byte boundary. Otherwise, the hardware traps (interrupts) into the alignment fault handler in the kernel which kills the process. We will revisit this architectural requirement to enhance an attack detection algorithm.

3 Implementation

The beauty of StackGhost is that it doesn't have to operate on every function call. StackGhost only needs to be invoked in deep function call sequences or recursive programs – when the program overflows or underflows the register windows and thus interact with the stack. If a program only performs shallow function call sequences, StackGhost may never be invoked to write register windows to the stack.

The hardware ultimately has the responsibility to decide when a register window needs to be written to or read from the stack. When the decision occurs, the processor automatically invokes the overflow or underflow handlers and StackGhost in the process.

To maximize the security afforded by StackGhost, the mechanism either needs to prevent a corrupt return address from exploiting oversights in the program or it needs to detect corruption of the return pointer. Unfortunately, one does not guarantee the other.

In order for a corrupt return pointer to exploit the system, the return pointer must actually be pointed carefully back into exploit code. A reversible transform can be applied to the legitimate return address and the result written to the process stack. When the return pointer needs to be accessed, the reverse transform can be applied before the access completes. Thus the value saved on the stack is actually a computation of the real return address. To retrieve the real value, the inverse computation is calculated. If an attacker does not know the transform or the key to the transform, he or she cannot predictably affect execution.

There are two ways to transparently detect a corrupt return pointer. The first is a function of the above transform function. Since the Sparc processor requires that all instructions be aligned on a 32bit word boundary, the lower two bits of an instruction's address will both be zero. The transform can invert one or both of the two least significant bits. A corrupt return pointer on the stack will be detected if those bits are not set at the time of the inverse transform. The Sparc processor will take care of this detection in hardware and cause an alignment trap.

A corrupt return pointer can also be detected by keeping a return-address stack. Every time a return pointer is saved to the program stack, the handler can keep another copy in memory not accessible to the userland process (a return-address stack). A corrupt return pointer is detected if a function tries to return to a location not saved in the return stack.

3.1 Per-Kernel XOR Cookie

The most trivial incarnation of return pointer protection consists of XORing a fixed cookie into the return pointer. XORing the cookie with the pointer before it is saved onto the stack and again after it is popped off preserves the legitimate pointer but will distort any attack.

By setting one or both of the two least significant

bits (LSBs) in the cookie, the XOR not only inhibits exploit, it also can detect a corrupt return pointer. If an attacker does not know which of the LSBs are set on the stack, the corrupt return pointer will cause an alignment fault unless the attacker gets lucky. Even if each state of the LSBs are tried, the interaction with the remainder of the cookie shound hinder the predictable operation of a corrupted return pointer.

A Per-Kernel XOR cookie can be built into OpenBSD by adding about a dozen assembler instructions. A sign-extended 13-bit cookie can be built into the kernel as an immediate operand that will cost one extra instruction per window pushed and another instruction when the window is popped.

The Per-Kernel XOR cookie can also be trivially defeated. It is constant for every process on the system. The cookie can be determined if an attacker is able to run arbitrary programs. Even without a priori knowledge of the cookie, an attacker could use a large shell code sled to slide into the main exploit code [1].

The Per-Kernel XOR cookie will not be enough to stop a competent attacker.

3.2 Per-Process XOR Cookie

A safer alternative to a per kernel XOR cookie would be to use a different random cookie for every process. The cookie can be stored in the Process Control Block (PCB) outside of user readable memory. The PCB will be automatically copied on a fork() and re-initialized on an execve(). There is even an extra 32-bit padding field in the OpenBSD PCB structure that can be used to store the 32-bit cookie.

A Per-Process XOR Cookie is far safer than per kernel granularity. But the XOR cookie can be inferred if an attacker can read the distorted return pointer off the stack and can also predict what the real return pointer should be. Format string vulnerabilities allow the first condition and looking at the vendor supplied application binary can often provide the real return pointer.

The Per-Process Cookie overhead will add approximately four instructions of overhead to both the push and pop action. In a few instances it will be as few two when the PCB pointer is already available in a register.

3.3 Encrypted Stack Frame

We can further mitigate detection of a corrupt return pointer with a more unpredictable transformation of the return pointer. We have the option of encrypting part of the stack frame when the window is written to the stack and decrypting it during retrieval.

Unfortunately there are several major obstacles to encrypting the return pointer.

1. Encrypting and decrypting every frame may seriously hinder performance.
2. At best, there are only 16 registers to work with. Auxiliary space would have to be statically allocated in the PCB.
3. The algorithm cannot rely on block chaining since userland threading or setjmp-longjmp could shuffle the call-return ordering.
4. The plaintext is easily predictable. Most of the high bits of the frame pointer will be set. Most of the high bits of the return pointer will be zero. The input registers (function arguments) will be fairly constant.

We believe a 64-bit block algorithm would offer improved security over the XOR cookie methods described above by using the concatenation of the frame pointer and return pointer as the input to the encryption algorithm. It could be a cryptographically **weak** usage but would stop all but the most determined adversaries. Encrypting the stack frames would unfortunately impose significant performance degradation for obvious reasons.

The encryption algorithm would have to be modified to encrypt the stack frame if StackGhost must detect a corrupted return pointer. The previous two StackGhost methods used the two LSBs as a form of an in-band secret. Using encryption as the transform would obviously cause the LSBs to be random.

3.4 Return-Address Stack

The pinnacle incarnation of StackGhost would implement what processor architects call a "return-address stack". To improve unconditional branch prediction, modern processors keep a FIFO stack in silicon of the return addresses of function calls [15, 11, 3]. Every time a *CALL* instruction is executed, its return address is pushed onto the stack. Every time a *RETURN* instruction enters the pipeline, the next address is popped off the stack and the processor continues fetching from the associated address seamlessly. A few cycles later, the real return address will be established and the processor can recover from a misprediction if need be.

We shall describe the theory developed to date. Our design criteria are as follows:

1. The mechanism must break no standard or common software.
2. The mechanism must guarantee the detection of a smashed stack.
3. The mechanism must kill any process with a corrupt stack.
4. The mechanism must have negligible memory utilization.
5. The mechanism must be implementable and debugable.

An obvious first approach might be to build in a return-address stack as a FIFO queue just as is done in hardware. Unfortunately, something so simple would break userland threading, setjmp() and longjmp(), and possibly C++ exceptions. Setjmp(), longjmp() and C++ exceptions introduce the problem that multiple return addresses can be bypassed by a deep multi-level return. This can be solved by scanning the entire stack until the return address can be located. Userland threading introduces a similar situation. When a thread relinquishes the processor to a sibling thread, it switches to a seperate stack. The second thread may be at the apex of a deep calling sequence and start returning. Again the queue will be out of order instead of FIFO and may have to be walked for every return. Performance will be sacrificed. If a thread is terminated or a program longjmp()'s, queue entries will reference stale stack frames and persist until the process terminates.

A more refined approach to designing a return-address stack is to add a small hash table in the PCB. Every time a register window needs to be cleansed, the mechanism would add an entry into the hash table (indexed off the base address of the stack frame). And then store the base address to use as the comparison tag, the return pointer, and a random 32-bit number. In the place of the return address in the stack frame, it would place a copy of the random number. When StackGhost retrieves the stack frame to refill the register window, it can compare the random number on the stack with its image in the hash table. If the instances do not match, an exploit has occurred and the program must be aborted. Otherwise, StackGhost fills the register return address with the one stored in the hash table.

A return-address hash table alleviates the performance problems associated with userland threading but does not address the memory leak associated with setjmp and longjmp or a terminated thread. Fortunately, setjmp and longjmp are both assisted by the kernel as a system call. Upon receiving the longjmp syscall, the kernel can walk backwards through the stack until the setjmp location is found, removing the hash entries along the way. An indirect benefit of walking the stack is that it also helps secure the jmpbuf (setjmp storage buffer).

For operating systems other than OpenBSD that support symmetric multiprocessing on Sparc and with kernel managed threads, mutual exclusion would have to be guaranteed at some level on the hash table. A locking primitive per window overflow and underflow handler invocation may prove prohibitivly expensive.

Further testing in a careful multi-user environment would be needed.

4 Performance Effects

4.1 Micro Benchmarks

Micro benchmarks were run under each of StackGhost's protection mechanisms and the results appear above in Figure 2 (see appendices for benchmark code and details). For the Return-Address stack mechanism, an optimistic approximation was implemented. It assumed an adequate number of pre-allocated entries its the free list and a naive

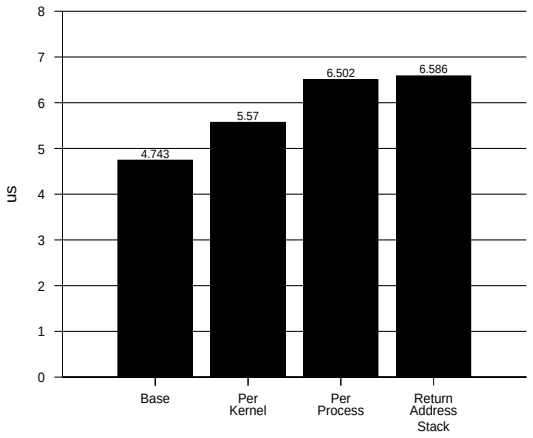

Figure 2: Microseconds per Function Call

random number generation scheme. Both Cookie methods are the true StackGhost implementations.

The micro benchmarks show a worst case scenario with a deeply recursive instance of an eight instruction function. Each of the function calls will invoke StackGhost. On a 70Mhz Sparc 4, the Per-Kernel XOR cookie imposes a little under one microsecond per function call penalty. The Per-Process cookie StackGhost overhead is a little under two microseconds per call. The return-address stack cost negligably more than the Per-Process mechanism.

In the **absolute** worst case (shortest possible recursive function that will still return), the Per-Kernel XOR cookie causes a 17.44% overhead over the baseline. The Per-Process XOR cookie can result in a 37.09% overhead. The return-address stack approximation imposes a 38.86% overhead.

Again, it **cannot** get worse unless there are unwieldy cache or TLB affects. We speculate that a bulk of the overhead is actually attributable to an additional TLB and cache miss instead of the additional instruction count.

The performance penalties could be reduced if the StackGhost code was interleaved into the trap handlers instead of just inserted. Sparc processors are superscalar, albeit in-order, and can take advantage of some instruction level parallelism (ILP). If the trap handlers themselves were re-written to increase ILP, the optimization should absorb most of the StackGhost cost.

4.2 Macro Benchmarks

The SPEC95 integer benchmark suite was also run to establish macro benchmarks (see Appendix 1 for environmental details). The results appear below in Figure 3.

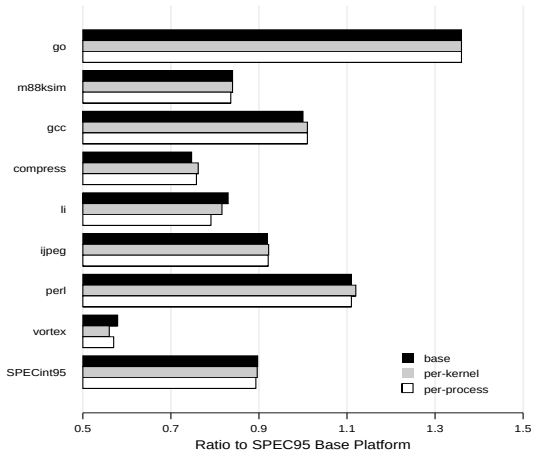

Figure 3: SPECint95 benchmarks

The performance penalties measured by the SPEC95 integer benchmark suite indicated that StackGhost only shaved a few hundreths of a point off the speed metrics. The performance aberrations may be more attributable to noise than the StackGhost mechanism. An un-StackGhosted OpenBSD 2.8 kernel had slightly worse performance than a StackGhosted version in some instances, the only other explanation is that it is due to cache or TLB effects.

Discounting any noise in the benchmark, the geometric mean SPEC rating showed a StackGhost overhead of 0.1% with a Per-Kernel cookie and a 0.4% overhead with a random Per-Process cookie.

5 Limitations

There are several moderate to serious deficiencies in StackGhost. Some could be diminished with further research but others are inherent.

5.1 Unpredictable Execution

The StackGhost XOR Cookie methods of hindering exploits do not always detect the corruption of the stack. If the attacker's return pointer manages to align correctly after being XORed with the cookie, execution will be transfered in an unpredictable manner.

Execution may divert to a random but legal stretch of code and cause data corruption. Of course, a successful attack may have the same chance of causing data corruption since no cleanup code will be called anyway.

5.2 Forked Processes

In the current StackGhost incarnation, forked processes have an identical Per-Process cookie.

It may be possible to unroll the stack and adjust each return pointer in the new process. But the process would have to be non-threaded and it would duplicate the program stacks instead of using a copy-on-write mechanism – potentially driving up memory utilization. Again, further research must be done.

5.3 Rootshell vs. DoS

If StackGhost saves a network daemon from a successful attack, it should abort the network daemon. A rootshell exploit will just be converted into a denial of service exploit since the daemon will be down. This behavior is an added incentive to remedy the underlying problems, instead of just mitigating exploit.

5.4 Random Pool Depletion

If random Per-Process keys are used, bursts of rapid program spawning could deplete the randomness pool. A starved pool could hinder other programs from executing until more randomness can be gathered.

5.5 Debuggers

Userland debuggers are currently broken by the XOR cookies. They will not be able to backtrace since the in-core return pointers are obviously distorted. The in-kernel core dump mechanism may be able to walk the stack and cleanse each activation record in the program. Further research must be done. Threaded programs would present an additional beast for reasons outlined above by the kernel return stack.

Debugging via Ptrace() will also present problems for the parent processes since the in-core program counter will have been modified by StackGhost.

5.6 Granularity

The current implementation of StackGhost protects each userland process on the system. It may be desireable to selectively protect processes deemed to be "at risk." Setuid, setgid and otherwise privileged processe are the likely candidates for automatic coverage. The XOR cookie mechanisms of StackGhost may disable coverage by using a NULL cookie since XORing any number with zero is the equivalent of adding by zero – no effect.

5.7 Unaffected Exploits

StackGhost will not stop every exploit, nor will it guarantee security. Exploits that StackGhost will **not** stop include:

1. A corrupted function pointer (atexit table, .dtors, etc.)
2. Data corruption leading to further insecure conditions.
3. "Somehow" overwriting a frame pointer with a frame pointer from far shallower in the calling sequence. It will short circuit backwards through a functions' callers and may skip vital security checks.

6 Other Architectures

The application of the StackGhost mechanisms to architectures other than Sparc is contingent on a trap into the kernel when registers are pushed onto the stack and again when they are popped off. We know of no common architectures which provide the convenience of the Sparc register window overflow and underflow traps. But we believe there are several architectural features which could approximate the behavior of Sparc.

6.1 Hardware Breakpoints

Many architectures provide hardware assisted debugging in some fashion. If the support comes in the form of hardware breakpoints, and the breakpoints can be placed on memory accesses (aka watchpoints) instead of instruction addresses, the kernel may be able to load a breakpoint on the address where the next and last return pointers are stored before context switching into a processes. Access to the current function's return pointer and the next function's in the calling sequence would both cause a trap into the kernel. A deeper function call will cause a trap which must add a breakpoint to the next return pointer location when it saves the current. A return will cause a trap which must confirm that there is a breakpoint on the next previous return pointer.

The use of hardware breakpoints to approximate the trap behavior of register windows requires knowledge of stack layout a priori or the userland process must include stack layout hints.

6.2 Page Protection

Most architectures allow some protection mechanism to limit access to virtual pages. By marking stack pages as unaccessible (for both reads and writes), the kernel could guarantee a trap every time the stack is accessed. Unfortunately, the kernel will be notified for every stack access. A variation of this method proved too costly on IA32 under the MemGuard implementation [4] briefly described later.

6.3 IA64

Intel's IA64 architecture supports variably sized register windows (the *Register Stack Engine* in the Intel vernacular) [7, 8, 9, 10]. In IA64, function's can request an arbitrary number of registers unlike the 24 register window on Sparc. If an overflow or an underflow occurs, the processor stalls while the hardware interacts with the backing store. The actual loading or storing of the registers is done by the hardware instead of by kernel trap handlers.

To simulate the Sparc register window trap behavior, it may be possible to misalign the backing store pointer. Every time the Register State Engine stores to the backing store or retreives registers from it, there will be a trap into the kernel thus invoking the StackGhost mechanism.

7 Related Work

There have been several prior research endevours against buffer overflows and to guard the stack. This is by no means an exhaustive list.

7.1 StackGuard

Crispin Cowan's StackGuard is a modified compiler which places canaries (the term canary can be used interchangeable with our use of the term cookie) around the return pointer in function prolog. A buffer overflow will modify the canary on its way to overwriting the adjacent return pointer. If the function epilog detects a dirty canary, it rightly infers that an exploit has occurred, it logs the exploit and it aborts the program [4].

StackGuard can also XOR a random canary into the return address in the function prolog and XOR the canary out in the epilog. This should cause an undetected corrupt return pointer to dump core instead of executing the exploit code.

Another technique called MemGuard was described in the same paper as StackGuard. MemGuard designates the return address on the stack of an x86 machine as a "quasi-invariant." It only allows a store to that memory location through the MemGuard API. This involved marking the entire stack

page read-only during function prolog, and unprotecting the page during the epilog. A special trap handler was installed in the kernel to emulate the writes to the stack locations near the return address that were unfortunate enough to fall on the same virtual memory page. MemGuard proved to impose an inordinate overhead.

7.2 StackShield

StackShield works as an assembler processor supported by the GNU C and C++ compilers. It works by modifying the function prolog to store away the return pointer into a stack distant enough that overflow is not likely. Upon function return, the function epilog actually returns from the location specified in the private return stack instead of the program stack [18]. The only exploit detection StackShield performs is checking the segments function pointers point to.

7.3 ProPolice

Hiroaki Etoh's ProPolice is a modification to the GNU C compiler that places a random canary between any stack allocated character buffers and the return pointer [5]. It then validates that the canary has not been dirtied by an overflowed buffer before the function returns. ProPolice can also reorder local variables to protect local pointers from being overwritten in a buffer overflow.

7.4 LibSafe

LibSafe is a library modification to Linux that safely wraps functions known to be "unsafe" and contains any damage to the local stack frame [2]. Also included in the LibSafe paper is a tool called LibVerify that will rewrite a binary application to perform a return address check.

7.5 Non-Exec pages

There are several implementations available that attempt to hinder an exploit by limiting the memory segments that code can execute in.

Solar Designer architected a kernel modification to x86 Linux to prevent execution in stack pages. Exploits will not be able to run their own code if the buffer resides on the stack (which most buffer overflows do) [16]. Sun also built an optionally enabled non-executable stack into the Sparc version of Solaris.

PaX is a x86 Linux kernel modification to mark all data pages non-executable, not just stack pages. PaX inhibits heap exploits in addition to stack overflows [14].

There are several overflow exploits that non-executable pages do not inhibit. By far the most common is the "return into libc." Instead of executing custom exploit code, the attack directs the return pointer back into code can have malicious consequences depending on its arguements. The easiest example is to point the return address back at the system() library call and point the arguement at an instance of "/bin/sh".

8 Conclusion

StackGhost has proven to be an effective defense against common exploit techniques at a negligible cost to the user. StackGhost's primary merit is that it is a kernel modification and does not require mass recompilation or the administrative headaches of selective protection. The current implementation of StackGhost is deficient in that it cannot guarantee the explicit detection of a stack exploit, it can only foil the operation of an exploit.

When the seperate return stack apparatus of StackGhost is fully implemented, StackGhost will offer guaranteed detection of the traditional buffer overflow at a fraction of the cost of the other available stack protection mechanisms.

9 Availability

The StackGhost project homepage can be found at http://stackghost.cerias.purdue.edu. The relevent patches to OpenBSD shall be placed in the Public Domain.

10 Acknowledgments

We would like to thank Rick Kennell and Florian Kerschbaum for technical advice. And to thank Susan Hazel for assistance with paper organization and editing.

References

[1] Aleph One. "Smashing The Stack For Fun And Profit." *Phrack*, 7(49), November 1996.

[2] Arash Baratloo, Timothy Tsai, and Navjot Singh. "Transparent Run-Time Defense Against Stack Smashing Attacks." *Proceedings of the USENIX Annual Technical Conference, June 2000.*

[3] C. F. Webb. "Subroutine call/return stack." *IBM Technical Disclosure Bulletin, 30(11)*, Apr. 1988.

[4] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle and Qian Zhang. "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," *Proceedings of the 7th USENIX Security Conference*, January 1998, San Antonio, TX.

[5] Hiroaki Etoh. "GCC extension for protecting applications from stack-smashing attacks." http://www.trl.ibm.co.jp/projects/security/ssp

[6] Immunix.org. "StackGuard Mechanism: Emsi's Vulnerability." http://immunix.org/StackGuard/emsi_vuln.html.

[7] Intel IA64 Architecture Software Developer's Manual. "Volume 1: IA-64 Application Architecture Revision 1.1." July 2000

[8] Intel IA64 Architecture Software Developer's Manual. "Volume 2: IA-64 System Architecture Revision 1.1." July 2000

[9] Intel IA64 Architecture Software Developer's Manual. "Volume 3: IA-64 Instruction Set Reference Revision 1.1." July 2000

[10] Intel IA64 Architecture Software Developer's Manual. "Volume 4: IA-64 Itanium processor Programmar's Guide Revision 1.1." July 2000

[11] Kevin Skadron, Pritpal S. Ahuja, Margaret Martonosi, Douglas W. Clark. "Improving Prediction for Procedure Returns with Return-Address-Stack Repair Mechanisms." *Proceedings of the 31st ACM/IEEE International Symposium on Microarchitecture,* Nov 1998, Dallas TX.

[12] Mudge. "How to Write Buffer Overflows." http://l0pht.com/advisories/bufero.html, 1997.

[13] Nathan P. Smith. "Stack Smashing vulnerabilities in the UNIX Operating System." http://millcomm.com/~nate/machines/security/stack-smashing/nate-buffer.ps, 1997.

[14] PaX Team. "NonExecutable Data Pages." http://pageexec.virtualave.net/pageexec.txt

[15] S. McMahan. Cyrix Corp. "Branch Processing unit with a return stack including repair using pointers from different pipe stages." U.S. Patent No. 5,706,491. Jan, 1998.

[16] Solar Designer. "NonExecutable User Stack." http://www.false.com/security/linux-stack/.

[17] SPARC International, Inc. "The SPARC Architecture Manual." Version 8. 1992.

[18] Vendicator. StackShield: A "stack smashing" technique protection tool for linux. http://www.angelfire.com/sk/stackshield/.

Appendix 1: Benchmark Procedure

A short C program was used to microbenchmark the function call overhead imposed by StackGhost. It was compiled with gcc version 2.95.3 19991030 (prerelease).

The Spec95 integer suite was run to generate the macro benchmarks. The benchmark suite was built with gcc version 2.95.3 19991030 (prerelease).

All the benchmarks were run on a 70Mhz SparcStation 4, 32MB of ram, PROM Rev 2.20, and no L2 cache. The machine was operating in multi-user mode under fairly constant conditions for each iteration of the benchmarks.

Appendix 2: Micro Benchmark

```
#define DEPTH 10000
#define TRIALS 1000

void deep(int n)
{
  if (--n)
    deep(n);
}

int main(void)
{
  struct timeval start, stop;
  float total, times[TRIALS];
  int i;

  /* Prefault the stack */
  deep(DEPTH);

  for (i = 0; i < TRIALS; i++) {
    usleep(1); /* Give up time slice to avoid context switch */
    gettimeofday(&start, NULL);
    deep(DEPTH);
    gettimeofday(&stop, NULL);

    times[i] = stop.tv_sec - start.tv_sec + (float)(stop.tv_usec - start.tv_usec) / (1000000);
  }

  for (i = 0, total = 0; i < TRIALS; i++)
    total += times[i];

  printf("Avg time %.5fs\n", total / (float)TRIALS);
  printf("Avg us/call %.3fus\n", (1000000 * total) / (float)(TRIALS * DEPTH));
}
```