StackGuard: Simple Stack Smash Protection for GCC

Perry Wagle *Crispin Cowan*[*]
Immunix, Inc.[†]
wagle@immunix.com, http://www.immunix.com/~wagle

Abstract

Since 1998, StackGuard patches to GCC have been used to protect entire distributions from stack smashing buffer overflows. Performance overhead and software compatibility issues have been minimal. In its history, the parts of GCC that StackGuard has operated in have twice changed enough to require complete overhauls of the StackGuard patch. Since StackGuard is a mature technology, even seeing re-implementations in other compilers, we propose that GCC adopt StackGuard as a standard feature. This paper describes our recent work to bring StackGuard fully up to date with current GCC, introduce architecture independence, and extend the protection of stack data structures, while keeping the StackGuard patch as small, simple, and modular as possible.

1 Introduction

Despite years of punditry, source code audits, and many layers of proposed technology, buffer overflows are *still* the leading cause of software vulnerability. This paper describes the motives and technical issues of incorporating the StackGuard [6] stack smash defense as a standard feature of GCC.

The *stack smashing* variety of buffer overflow [14] is its most common subtype, and the most readily treatable. A stack smash attack gains control of a thread in an address space by overwriting control information—such as a return address—on its stack.

The common way for the attacker to overwrite values stored on the stack is to use a *buffer overflow*, where large inputs are used to cause more data to be written to an area of memory than space has been allocated. StackGuard protects against stack smash attacks resulting from buffer overflows, but also those resulting from *any* sequential write through memory.

To detect corrupted control information in procedure activation records, StackGuard adds a location that it calls a "canary"[1] to the stack layout to hold a special guard value. Traditionally, the layout of that section of the stack has been determined by function *prologue* and *epilogue* code generators, which are architecture specific. As a result, StackGuard *implementations* have also been architecture specific. As time has passed, these parts of GCC have become more abstract, requiring repeated re-implementations of StackGuard, but the ability to modify stack layout in a platform independent way has been lacking.

A new version of StackGuard has been imple-

[*]This work supported by DARPA Contract F30602-01-C-0172.

[†]nee WireX Communications, Inc.

[1]Alluding to the canary Welsh miners used to detect air problems before the miners could.

mented for modern versions of GCC. It now guards *all* the information in the control region of all procedure activation records generated by the `./gcc` back end of the GCC compiler suite (C, C++, *etc.*). That is, the saved registers and saved frame pointer are are now protected in addition to the return address for every procedure. Stack layout to provide the canary *location* is still left to the architecture specific function prologue and epilogue code generators. The rest of StackGuard is now architecture independent.

This new StackGuard has been successfully used in conjunction with other security hardening technologies to rebuild the Red Hat 7.3 distribution (GCC 2.96-113). The StackGuard patch has also been applied to the source for GCC 3.2-7 used in the Red Hat 8 distribution to rebuild both the compiler and GLIBC.

In accordance with the principle of default deny [15] StackGuard makes a point to apply the guarding technology to *every* procedure in a distribution. In this fashion, StackGuard is a *security optimization* that transforms *all* emitted code to deny a class of attacks. It also shows the soundness of the transformation by showing that the distribution works the same after the transformation as it did before. Picking and choosing which procedures receive the transformation is a *performance optimization* that, based on *assumptions* about the nature of the security threat, trades some security for performance.

StackGuard strives to be the essence of the "guard the control information" security optimization that is capable of being applied to every procedure on a system. Given the negligible performance impact of the complete transformation [5, 7], we have never seen the need to apply any additional performance optimizations. Unfortunately, there persists the mistaken impression that StackGuard produces a significant performance impact [1].[2]

This paper describes the issues to be considered for including StackGuard as a standard feature in GCC. In keeping with good modular design principles, we emphasize the considerations specific to the stack smash detection technique to keep that problem small and separate. In particular, the guarding technology can be used, with compiler support, to guard other regions of memory. Its design and implementation should not be unnecessarily tied to the specific use as a Stack Smash detector, though that's all that is discussed in this paper.

The rest of this paper is as follows. Section 2 describes compiler work to date on the buffer overflow problem. Section 3 describes the design of our proposed feature for GCC. Section 4 describes our current implementation of this design in GCC 3.2. Section 5 presents our performance benchmarks, supporting our claim that this feature is low-cost. Section 6 describes our on-going security testing. Section 7 presents our conclusion. Section 8 describes the availability of the StackGuard technology.

2 Background and related work

Aleph One's paper [14] presented a cook book for the "stack smashing" variety of buffer overflows, in which the attacker overflows a stack buffer to change the return address in an activation record to point to malicious code contained in that self same overflow. In the general case, the attacker wants to inject malicous code, and alter control flow structures so that the program will jump to the malicious code. The stack smash is an elegant attack that

[2] The performance issues shown in the Libsafe paper result from selected benchmarks (quicksort) that emphasize where StackGuard imposes overhead (function calls) and ignores where Libsafe imposes overhead (string functions).

achieves both objectives in a single stroke by exploting a very common programming error (weak bounds checking on fixed sized stack buffers). However, the attacker only *needs* to change control flow, because subvertable code (capable of performing the moral equivalent of "`exec(sh)`" for the attacker) is often already resident in the victim program's address space.

Since Aleph One's paper appeared, there has been a lot of work to defend against buffer overflows, interceding in the operating system [9, 8, 11, 17], system libraries [16, 1], and compilers [6, 12, 13, 10, 19]. These techniques variously try to prevent the modification of control flow paths, prevent the injection of malicious code, or both [7].

Because we are proposing a GCC enhancement, we consider only compiler defenses. Compiler defenses can in turn be divided into array bounds checking (which prevents buffer overflows, described in Section 2.1) and data integrity checking (which detects buffer overflows in time to prevent attacks from succeeding, described in Sections 2.2).

2.1 Bounds Checking

Array bounds checking is the ultimate way to eliminate buffer overflows. Unfortunately, the design and idioms of the C language make it difficult to provide for fully secure array bounds checking while preserving reasonable legacy compatibility and reasonable performance.

The Compaq C compiler for Tru64 UNIX [3] is an example of incomplete protection. The compiler has an option to perform bounds checking, but it only does so on *explicit* array references; pointer references are not checked. Since all arrays passed as function arguments are converted to pointers, this means that array bounds checking is effectively limited to strictly local variables, and the security value of the feature is low.

The Jones and Kelly GCC enhancement [12, 18] is an example of compromised performance. This GCC enhancement provides complete array bounds checking, even for pointer references, and maintains the current size of a pointer as a machine word. They achieve this through an associative lookup on each pointer reference to an array descriptor that stores the base and bounds. Performance penalties are high, approximately 10X to 30X slowdown.

The Bounded Pointers project [13] is an example of compromised compatibility. Rather than associative lookup, Bounded Pointers changes pointers from a single word into a tuple that incorporates base and bounds. This improves performance by eliminating the associative lookup in Jones and Kelly, but also costs compatibility because pointers no longer fit in a single word. Performance penalties are still high at approximately 5X slowdown. It is conjectured that this slowdown could be substantially reduced, but unlikely that the penalty would reach the low percent range.

2.2 Integrity Checking

The first integrity checking mechanism was Snarski's libc [16] that checked the integrity of activation records within libc functions. StackGuard [6] generalized this notion with a compiler enhancement to check the integrity of *all* activation records. These methods ornament activation records as they are built with data structures that cannot survive stack smashing attacks, so that when the function tries to return, it can detect that the activation record has been corrupted. Upon detection, the program issues an intrusion attempt alert and exits, rather than handing control to the attacker.

There have been three major releases of Stack-

Guard. StackGuard 1 was a patch to GCC 2.72, hooking directly into the `prologue` and `epilogue` code generation functions to emit StackGuard canary generation and verification code into function set up and tear down. StackGuard 2 was a complete re-write, providing an enhancement to GCC 2.92, this time implemented as modified RTL generation for function setup and tear down. StackGuard 3, presented here, is another complete re-write to accomodate GCC 2.95 and newer.

There are two significant reimplementations of StackGuard: Propolice and Visual C++.net.

OpenBSD's Propolice implements something very much like the StackGuard defense as an enhancement to GCC, and provides a important and very interesting contrast with its differences:

- To a large extent, Propolice and StackGuard have independently converged, from opposite directions, on doing the guarding code inserts at the RTL level.[3] But they haven't quite met in the middle—Propolice does the inserts at a much earlier pass in the compiler.
- Propolice places the canary word, as a *buffer overflow* detector, *only* at the top of auto variable regions containing "buffers".[4] StackGuard places the canary word, as a *stack smash* detector, at the bottom of *every* control region.
- Propolice uses random canaries. StackGuard uses terminator canaries.
- Propolice provides variable sorting—moving *some* character arrays above all other data types—to make it difficult to overflow into *adjacent* variables. StackGuard makes no assumptions about the starting point of runaway sequential overwrites of the stack, leaving security optimized stack layouts to separate mechanisms, such as Propolice.
- Propolice appears to move significantly towards a universal, architecture independent, stack layout. It even goes as far as to move saved registers into the autovariable region. StackGuard goes to great pains to try to leave the stack layout as close to the way it was as possible.
- Propolice modifies far older versions of GCC than StackGuard.[5]

Propolice's design decisions present different trade-offs than StackGuard:

- By doing the code inserts well before sibling and tail recursion is recognized, Propolice has no way to insert canary checks before the function exits points produced by the external branches. StackGuard makes a point of doing these insertions also.
- By depending on the *coincidental* adjacency of the autovariable region and the control region on the stack, Propolice gives the appearance of guarding the control region from buffer overflows that it detects leaving the autovariable region. But this *implicit* invariant isn't maintained across compositions with other security and performance oriented transformations that affect stack layout.
- The apparent strengths and weaknesses of both terminator and random canaries are discussed in Section 3.1.

[3] Propolice started at the AST level, while StackGuard started at the architecture specific function prologue and epilogue backend level.

[4] Currently, this appears to be defined as character arrays of greater than 4.

[5] OpenBSD's GCC 2.95.3 20010125 *vs.* Redhat 7.3's GCC 2.96-113 and Redhat 8.0's GCC 3.2.2-2.

- Nothing requires string writes to start in a char buffer. When an exploit finds such an opportunity, Propolice will stop it only if it's lucky. StackGuard will stop it by its design that *all* stack smashes should be detected.

- Propolice's buffer overflow detector becomes quite different than StackGuard's stack smash detector when alternate stack layouts, involving multiple stacks, stacks growing upward, heap allocated stacks, *etc*. are considered. Both are useful: Propolice detects buffer overflows that aren't stack smashes, and StackGuard detects stack smashes that aren't buffer overflows.

- By moving saved registers into the autovariable region, Propolice appears to assume that saved registers have the same dynamic scope rules as autovariables. This is not necessarily true for tail calls, where it would be correct to restore saved registers, but not correct, in general, to deallocate autovariables.[6]

- Propolice's changing of the stack layout could disrupt other tools that do stack introspection, such as GDB and JIT-styled JVM's.[7] StackGuard goes to pains to be invisible to such tools.

- It's unknown how well Propolice ports to current versions of GCC. StackGuard strives to be its part of that work, done completely and correctly.

Microsoft has also implemented [4] a feature very similar to StackGuard which they call the "/gs" feature in Visual C++.net. Compiler implementation details are naturally closed source, but the emitted code strongly resembles StackGuard code. The comparison is gone into more detail at various points in later sections.

Section 3 presents the StackGuard 3 design in more detail.

3 Design

The purpose of StackGuard is to do integrity checking on activation records, with sufficient precision and timeliness that a program will never dereference corrupted control information in an activation record, which is written to once on entry to a function, and read from once on exit from a function.

The *threat* is that the attacker has the capability to overwrite control information in some frame on the stack via a sequential write operation—such as a string copy or a memory copy—starting from somewhere lower in memory than the *target*. This permits the attacker to hijack the thread to execute code of the attacker's choice. The desired code might be new code supplied by this particular sequential write into the stack, another sequential write into stack *or non-stack* memory, or else code that is already in the address space that will do what the attacker needs when branched to in this fashion.

We will assume that the attacker does not need to inject code, but can use executable code already in the address space. This is a growing technique in practice, and permits us to focus on the most important part of the attack: overwriting control information, particularly pointers to code, such as return addresses.

The attack works if it can rewrite the control information between the time it was written with correct values to be saved and the time it was later read assuming it contained correct values of things to be restored.

[6] Some really clever tricks would be needed to support tail recursion from functions with autovariables, and people have been known to build compilers that do that.

[7] As has happened with previous (but not current) versions of StackGuard.

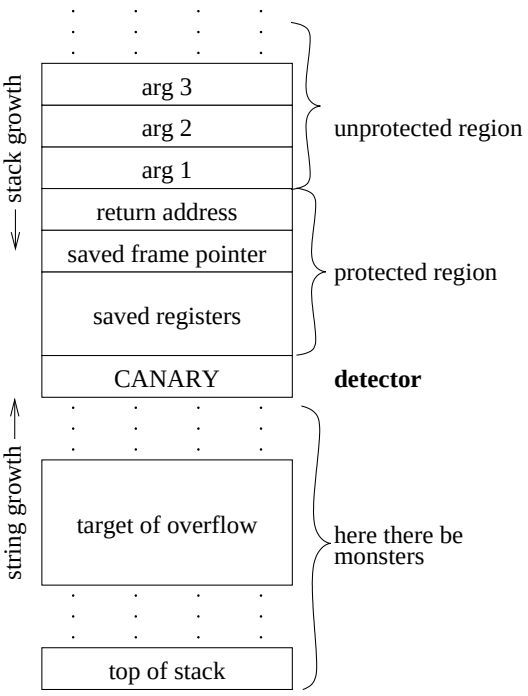

Figure 1: i386 Stack Layout

The *defense* is to insert a **canary location** immediately before the control information in each frame on the stack. See Figure 1. Any sequential write through memory, such as by a buffer overflow, that tries to rewrite the control information will be forced to also rewrite the canary location. Then the remaining problem is to make the value of the canary something that's hard to spoof. The canary location is **initialized** immediately after the control values are saved, and **checked** immediately before the control values are restored.

The control region is protected by virtue of the fact that the canary is checked before each use of the protected information. The arguments sitting above that are used sooner than that, so aren't protected.

3.1 Types of Canaries

There are three kinds of canaries, each with a different strength and weakness:

- **terminator canaries** detect runaway strings, but is a known value.
- **random canaries** detect all sequential memory writes that don't know its secret value.
- **random XOR canaries** are random canaries that might also detect random-access memory writes into the protected region.

Terminator canaries leverage the observation that most *stack* buffer overflows involve string operations, and not the memory copy operations almost always applied instead to heap allocated objects, by using a value composed of four different string terminators (CR, LF, Null, and -1). The attacker can't write the terminator character sequence for the particular string operation being used to memory and then continue writing, because one or more of the terminator characters halt the string operation.

If the exploit gets to overwrite the canary more than once, it can overwrite the protected control information on the first write, and then reconstruct the canary value with consecutive writes. It's not known how rare multiple write vulnerabilities are.

Any memory copy will be able to write the terminator canary value.

Random canaries assume that the exploit can sequentially write any value it wants and keep going. So it forces the exploit to know a 32-bit secret random number thats retrieved from a global variable that's initialized to a different value each time the program is executed. Memory protection techniques can be used to protect the global from writing, such as isolating the global on its own page and bracketing it with "red" (unmapped) pages. The exploit might also be able to get the victim program to tell it what the current random canary value is,

having it read from either the stack or from the global.

If the attacker can also deploy an exploit that can read the random canary value from anyplace it might also reside in memory, then both string and memory copies can easily overwrite the correct value (unless it contains the appropriate string termination characters, and thus less entropy).

Random XOR canaries assume that the exploit might be able to random-access write to the location of some of the protected information [2]. So in addition to employing the random canary defense, some or all of the saved control information is exclusive-or "encrypted"[8] with the random canary value, storing the result in the canary location. Then to change the protected control information the attacker needs to deploy an exploit that sets the canary location to the exclusive-or of the random canary value and the new values of the control values used in the full "encryption."

Random XOR canaries have the same weakness as random canaries above.

3.2 Examples of Canaries

All versions of StackGuard have provided terminator canaries. We know of no alternate implementations that provide this type of canary.

All versions of StackGuard up to, but excluding the latest version, provide random canaries. Propolice provides random canaries.

Only the mid-1999 version of StackGuard provided random XOR canaries, protecting only the region containing the return address. When we checked in early 2003, Visual C++.net's `/gs` option performed exactly the same algorithm with the return address, but positioned the canary to also protect (without the "encryption") the saved frame pointer.

[8] Some people are uncomfortable with the use of the word "encryption" to protect integrity instead of confidentiality, hence the scare quotes.

4 Implementation

The first thing a function does on entry is to save the caller's control information on the stack. The region of memory used for this must be protected by a canary location, which is initialized with the desired canary value.

The last thing a function does on exit is to restore the caller's control information from the region of memory set aside for that. But, before that restoration can take place, the canary value must be checked to see if it has changed. If it has, the stack has been corrupted, and the process is killed after a suitable Intrusion Response System has been notified.

The code generators are:

- **determine canary location**—decide on where the canary location is going to be on the stack and how the below operations are going to refer to it.
- **allocate canary location**—make space for the canary in the stack layout in a memory location close to and preceeding the region containing the saved control values to be protected.
- **initialize canary**—give the canary location its correct value before any operation happens that could rewrite it or its protected region of memory.
- **check canary location**—check that the canary location contains its correct value, after any operation happens that could rewrite it or its protected region, but before the saved control information it protects is restored with corrupted values. If

the check fails, invoke the fail stop operation below.

- **deallocate canary location**—remove the space made for the canary on the stack by the allocate canary location operation above.
- **perform fail stop**—send the mangled name[9], the type of canary, the correct value of the canary, and the corrupted new value of the canary to a security fault handler.

The traditional way to implement StackGuard has been to modify the function prologue and epilogue code generators, which are responsible for causing the machine instructions to be emitted that save the caller's frame pointer at the beginning of the frame (if frame pointers are enabled), saving registers, establishing the position independent code pointer if it is enabled, and possibly aligning the stack pointer to some boundary.[10]

We've decided on terminator canaries on the basis of the observation that nearly all *stack* overflows are via string operations.

It should be noted that the order of the application of the code generators is different than the order that the emitted code appears in the generated function. In particular, first, the body of the function is converted to RTL. Then a number of optimizations take place, until the sibling and tail recursion optimization makes its decisions available. Then the **initialize canary location** operation is added to the beginning, and the **check canary location** operation is added to all the exit points. Then some more RTL optimizations are performed until the function prologue adds code to the beginning and invokes the **allocate canary location** code generator, and the function epilogue adds code to all the exit points and invokes the **deallocate canary location** code generator.

In this paper, we try to keep a clear distinction between "code generators" and "operations." Code generators might be invoked in an arbitrary order to emit operations that appear in a desired order in the object code.

In an earlier section, we were critical of Propolice's *design*. In the remainder of this section, despite that it really does successfully recompile practically all of the Redhat 7.3 distribution for a production quality distribution, we are critical of StackGuard's *implementation* for the remainder of this section.

4.1 Determine Canary Location

The **determine canary location** code generator is architecture *specific*, since it needs to know how the stack is laid out.

Both the **initialize canary location** and **check canary location** code generators need the architecture specific RTX for referring to the location of the canary. But both are invoked before the architecture specific **allocate canary location** and **deallocate canary location** code generators are. It turned out that the i386 backend placed alignment padding (`padding1`) right where the soft frame pointer points:

```
rtx canary_loc
  = gen_rtx_MEM (SImode,
      frame_pointer_rtx);
```

and that was a nice location for the canary.

Marking the location volatile:

```
MEM_VOLATILE_P (canary_loc) = 1;
```

was required for the **initialize canary location**

[9] The variable containing the unmangled name isn't always initialized at the time it is needed.

[10] The author believes his IA-32 bias here merely adds concreteness to his examples, and doesn't build in bad assumptions.

code generator to keep its RTL from floating past things that could corrupt the canary.

But, it broke the GCSE pass of the optimizer for the **check canary location** code generator, apparently due to the way the infinite loop in the **perform fail stop** was constructed, and it appears not to be required to keep the RTL sufficiently pinned down.

4.2 Allocate Canary Location

The **allocate canary location** code generator is architecture *specific*, since it runs in the architecture specific function prologue code generator.

For i386, it turns out to be very simple. Currently, the i386 architecture's `ix86_compute_frame_size` function does alignment padding between the autovariable region and the saved control information region. The solution is to add another alignment to `padding1` if it's not big enough to hold the padding.

Since the padding is allocated when the stack pointer is decremented (stack grows downward) to also allocate the autovariables, the allocation has no performance impact at runtime.

4.3 Initialize Canary Location

The **initialize canary location** code generator is architecture *independent*, since it just inserts an assignment into the RTL of the current function immediately after the sibling and tail recursion recognition optimization:

```
emit_move_insn (canary_loc,
 GEN_INT(terminator_canary_host_value));
```

where the canary value happens to be a simple expression[11] not requiring evaluation as an expression:

```
static const int
 terminator_canary_host_value
 = 0x000aff0d;
```

if it wasn't such a simple expression, then you would need to worry more about its temporaries being spilled to the stack *where they can be attacked*. Random and random XOR canary value expressions are largely non-simple, especially when being compiled for position independent code.

The above RTL sequence is inserted before the first non-NOTE RTL in the current function. As remarked in section 4.1 above, designating `canary_loc` as volatile appears to be sufficient to keep the it from floating past something that could corrupt the protected control information, but this isn't very comfortable. I've been hoping to stumble on a good way to insert barriers in the RTL instead of depending on volatile.

Sometimes the machine instruction generated needs a register, and usually it doesn't. Thus the late insertion might confuse late stages of register allocation depending on information from stages earlier than the insertion.

4.4 Check Canary Location

The **check canary location** code generator is architecture *independent*, since it just inserts a conditional branch into the RTL of the current function immediately after the sibling and tail recursion recognition optimization (see the discussion in subsection 4.3 above):

```
emit_cmp_and_jump_insns
```

[11] There are two different sorts of simplicity, one at the source code level (see the talk on TreeSSA), and one at the RTL to ASM conversion level.

```
(canary_loc,
  GEN_INT(terminator_canary_host_value),
  /* comparison = */ comparison, /* EQ/NE */
  /* size       = */ 0,
  /* mode       = */ SImode,
  /* unsignedp  = */ 0,
  /* label      = */ else_label);
```

which is appended to the end of each function, and inserted before each tail-call.

If the expression for the canary is not simple, then you need to make sure that it doesn't grab corruptable temporaries from the same computation in the **initializer canary location** code at the beginning of the function.

At the end of each function, the comparison argument is EQ, because the test is used to branch around the **perform fail stop** code whose generation is described in the section 4.6 below. The `else_label` label jumped to in that case is refers to the normal function epilogue code that hasn't been generated yet.

Before each tail call, the comparison argument is NE, because the branch is to the **perform fail stop** code appended to the end of the function.

Dead code removal works correctly for all of these inserts.

The branch prediction of the EQ case appears to get flipped correctly to put the return on the fast-path.

The machine instruction seems to usually need a register, but sometimes not. The late insertion of this RTL when it needs a register may be causing the code generation in the "getdents" function in GLIBC which has the attribute `((regparm(3), stdcall))` to go awry in the GREG (global register allocation) pass.

4.5 Deallocate Canary Location

The **deallocate canary location** code generator is architecture *specific*, since it runs in the architecture specific function prologue code generator.

For i386, it turns out to be absolutely free. The alignment padding where the location resides is stripped off at the same time as the autovariables.

4.6 Perform Fail Stop

The **perform fail stop** code generator is architecture *independent*, since it mostly just inserts a call to an external function named "`__canary_death_handler`" using the GCC's internal `emit_library_call` procedure.

The `__canary_death_handler` is invoked with information such as the current procedure name, the version of stackguard, the type of canary, what the canary value was supposed to be, and what the canary is now that it has been corrupted.

It's not expected that recovery is possible from a corrupted stack, so if the `__canary_death_handler` returns control from its call, something is very wrong, and the only thing reliable to do is go into an infinite loop. The correct way to recover would be to setup a different stack that returns control to different code.

Exception handling does not work here, since the stack is corrupt. If you like, you might consider this to be a security *fault* as opposed to an exception.

The late insertion of the `emit_library_call` into the RTL might be causing trouble.

4.7 Summary of problems

Moving the RTL code generators for **initialize canary location** and **check canary location** out of the function prologue and epiloque code generators was essential for two reasons. First, the prologue and epilogue were invoked too late to be able to generate the desired RTL. Second, the prologue and epilogue are architecture specific, and architecture independence is highly desired.

However, the movement of these two generators appeared to be blocked in two ways. First, they appeared to only work correctly around the time of the sibling and tail recursion optimization pass. Second, this was fortuituous because this was also the first point where the insert points became available for adding **check canary location** immediately before function exiting branches (that is, before return statements and tail calls).

Ideally the movement of these two generators should proceed to the point that AST is converted to RTL (which would also fix any problems the call to `emit_library_call` might be causing), but that implies that sibling and tail recursion recognition also move to that point.

4.8 Debuggers, Exception Handlers, and Other Stack Crawlers

Previous versions of StackGuard placed the canary location immediately before the return address on the stack. This was quite confusing to programs that did their own ad hoc parsing of the stack, such as GDB, Mozilla's module loading mechanism, and IBM's Java JIT compiler.

All of these became non-problems with the latest version of StackGuard, which places the canary location in a spot where nothing's supposed to be.

The aspell packages for Red Hat 7.3 has a complex enough class system for handling "file not found" exceptions that something throws it off, and it runs off the top of the stack without finding an exception handler, and abort()'s. This appears to be a problem with a dwarf annotation interaction with the old exception handler in GCC 2.96-113, which would probably be fixed (or at least completely different) in current GCC.

Exception handlers should check canaries for each frame as they crawl up the stack soas not to use corrupted information. We're hoping to add such support to the new exception handler in GCC 3.x, just as soon as a distribution that we can build, strenuously test, and release uses it.

4.9 Testing

The assembly output of the StackGuard compiler has been inspected for correct output for many optimization levels, with and without frame pointers, PIC and non-PIC, inlines, and nested function declarations.

A parser of the disassembler output for the StackGuarded version of the main glibc library libc.so.6 was done. Every procedure was correctly StackGuarded, and several tens of tail-call sites were observed.

Previous versions of StackGuard rebuilt Red Hat Linux 5.1, 5.2, 6.0, 6.1, and 7.0. The current version of StackGuard has rebuilt Red Hat 7.2 and 7.3, with a rebuild of Red Hat 8 in progress.

The `getdents` function in GLIBC in the Redhat 8 rebuild has problems. It looks like the late insertion of the canary check causes GREG optimization phase to drive something insane enough to apparently be confused about the

sizes of various types. The RTL for the function suddenly becomes quite different starting about halfway through the function after that pass, with tremendous movement of temporary and register initializations.

5 Performance Benchmarks

Formal performance benchmarks are currently under way, but were not complete at press time. Previous performance benchmarks on StackGaurd 2 [5, 7] show very marginal overhead on real loads, especially those programs that actually face network attack. In particular, benchmarks of Apache loaded by webstone, and throughput benchmarks of OpenSSH through the loopback interface, show overhead that is within measurement noise: `http://immunix.org/StackGuard/performance.html`. We expect similar performance from StackGuard 3.

6 Security Benchmarks

Security testing (like total correctness) is always problematic, because you cannot test for security, you can only detect *vulnerability*. As in performance, security testing is still under way at press time. Past security testing of StackGuard [6, 5, 7] shows that StackGuard is effective in its narrow goal of stopping classic stack smashing attacks. Furthermore, unlike some of the kernel-based defenses [8] when StackGuard stops a vulnerability, it is *stopped*, i.e. revising of the attack code does not result in bypassing StackGuard protection.

The major exception to this claim is that some exploits can attack the frame pointer, which was left unprotected in StackGuard 1 and 2. StackGuard 3 fixes this by moving the canary below the frame pointer.

7 Conclusion

StackGuard is a very modest sized patch, with modest performance and legacy compatibility costs, and yet solves a very large problem: chronic stack smashing buffer overflows. Despite having been first innovated in GCC [6], Microsoft has implemented a StackGuard-like feature [4] *as a standard feature* ahead of GCC. We propose that it would be beneficial for the GCC user community if the StackGuard security optimization became a standard compile option in GCC.

8 Availability

StackGuard has always been distributed under the GPL, and is currently available at `http://immunix.org/stackguard.html`.

Copyright assignment to the FSF for the StackGuard patches is in progress.

References

[1] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent Run-Time Defense Against Stack Smashing Attacks. In *2000 USENIX Annual Technical Conference*, San Diego, CA, June 18-23 2000.

[2] “Bulba” and ”Kil3r”. Bypassing stackguard and stackshield. *Phrack*, 10(56), May 2000.

[3] Compaq. `ccc` C Compiler for Linux. `http://www.unix.digital.com/linux/compaq_c/`, 1999.

[4] Crispin Cowan. Re: In response to alleged vulnerabilities in Microsoft Visual C++ security checks feature. `http://online.securityfocus.com/archive/1/256416`, February 14 2002. Bugtraq.

[5] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Erik Walthinsen. Protecting Systems from Stack Smashing Attacks with StackGuard. In *Linux Expo*, Raleigh, NC, May 1999.

[6] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *7th USENIX Security Conference*, pages 63–77, San Antonio, TX, January 1998.

[7] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. In *DARPA Information Survivability Conference and Expo (DISCEX)*, January 2000. Also presented as an invited talk at SANS 2000, March 23-26, 2000, Orlando, FL, `http://schafercorp-ballston.com/discex`.

[8] "Solar Designer". Non-Executable User Stack. `http://www.openwall.com/linux/`.

[9] Casper Dik. Non-Executable Stack for Solaris. Posting to `comp.security.unix`, January 2 1997.

[10] Hiroaki Etoh. GCC extension for protecting applications from stack-smashing attacks. `http://www.trl.ibm.com/projects/security/ssp/`, November 21 2000.

[11] Mike Frantzen and Mike Shuey. StackGhost: Hardware Facilitated Stack Protection. In *USENIX Security Symposium*, Washington, DC, August 2001.

[12] Richard Jones and Paul Kelly. Bounds Checking for C. `http://www-ala.doc.ic.ac.uk/~phjk/BoundsChecking.html`, July 1995.

[13] Greg McGary. Bounds Checking in C & C++ Using Bounded Pointers. `http://gcc.gnu.org/projects/bp/main.html`, 2000.

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

[15] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9), November 1975.

[16] Alexander Snarskii. FreeBSD Stack Integrity Patch. `ftp://ftp.lucky.net/pub/unix/local/libc-letter`, 1997.

[17] 'The PaX Team'. PaX. `http://pageexec.virtualave.net/`, May 2003.

[18] Herman ten Brugge. Bounds Checking C Compiler. `http://web.inter.NL.net/hcc/Haj.Ten.Brugge/`, 1998.

[19] "Vendicator". Stack Shield. `http://www.angelfire.com/sk/stackshield/`, January 7 2000.