StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks

Will Henderson, Josh Dall'Acqua

COMP 310, McGill University

February 16, 2024

The Morris Worm

In 1988, Cornell graduate student Robert Morris released the first mainstream **computer worm** to propagate through the Internet, infecting an estimated 10% of UNIX systems connected at the time.

Though intending to be merely an experiment of how far the worm could spread, it effectively took down a good handful of computers.

Morris's Mistake

Morris knew that when he checked if a computer was already infected with the worm, some people may send false positives back. To get around this and keep the worm propagating, he tacked on a **14%** chance of infecting a computer that sent a positive signal.

Morris's Mistake

Morris knew that when he checked if a computer was already infected with the worm, some people may send false positives back. To get around this and keep the worm propagating, he tacked on a **14%** chance of infecting a computer that sent a positive signal.

Unfortunately, this meant some computers got infected many, many times...

The Morris Worm

This was a big deal. It caused monetary damages, lost people's trust in the Internet, and led to a conviction for Morris.

The Morris Worm

This was a big deal. It caused monetary damages, lost people's trust in the Internet, and led to a conviction for Morris.

…How exactly did he do it, though?

The Morris Worm

One of the main exploits he used was **buffer overflows**.

This presentation introduces the buffer overflow exploit and *StackGuard*, a 1998 paper that details a way to patch it.

Table of Contents

Table of Contents

Problem

- Buffer overflow vulnerabilities are common in legacy code
- It is easy to patch a vulnerability but expensive to find them
- What is a low cost, general way to prevent these vulnerabilities?

Goal

- Stack Guard aims to provide a low cost, effective solution to prevent buffer overflow attacks
  - Accomplished through a compiler patch
  - Does not modify source code
  - Focuses on preventing changes to a program's return address.

Ramifications

- Deescalate urgency in patching buffer overflow vulnerabilities
- Immediately increase confidence in security of existing code

Difficulties

- Adding detection of buffer overflow attacks is expensive
- Implementation must be general and work with existing code

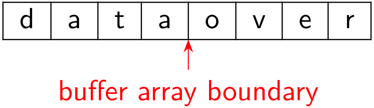
Table of Contents

Connection to Course Content

- Pages and Virtual Pages
- Layout of the stack in memory

What is a Buffer Overflow Attack?

- C and C++ **do not** provide standard array bounds checking
- Data can be written into memory after the end of the array (potentially outside of the program's memory block!)
- Exploitation opportunity: code can be injected into memory after the array



Stack Smashing Attacks

- A very common overflow attack is the ingeniously simple **stack smashing attack**, performed by providing an input string that works in two parts:
  - Malicious binary code that is executable on the machine being attacked, i.e., attack code, is injected onto the stack beyond the buffer, and
  - The return address of the function, also contained in a stack frame beyond the buffer, is modified to then jump to the attack code.
- Most often, this type of attack is used against privileged daemons to create a new `root` shell, providing the attacker with `root` privileges.

Table of Contents

How Does Stack Guard Work?

- Prevent injected code from changing the function's return address
- Stops running program (potentially on a privileged daemon) from entering injected attack code.
- Two approaches with different performance trade-offs
  - Canary word
  - Memory Guard
- Both approaches modify `gcc` to check the integrity of the function's return address before returning

Canary Stack Guard

- Place a "canary" word next to the return address on the stack

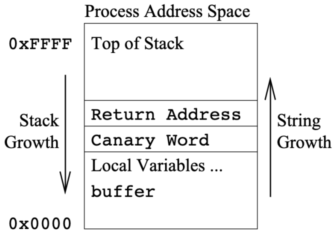


Figure: Canary word location in stack, from [1].

- If the attack code attempts to change the return address via overflow, the overflow will overwrite the canary

Canary Stack Guard

- Before returning, the function epilogue checks that the canary word is intact
- Randomizing the canary
  - A constant canary is easy to guess depending on how much access the attacker has
  - It is then easy to bypass Stack Guard
  - This can be avoided by randomly choosing a canary word

Virtual Memory Mem Guard

- Protect function return address when the function is called and un-protect it when the function returns
- The page in virtual memory where the return address is stored must be marked read-only
- Mem Guard API will emulate writes to non protected words on the page while keeping the return address protected
- Comes with a larger performance penalty than the canary word approach
  - Emulating Writes to non protected words is very expensive

Pentium Debug Register Mem Guard

- Pentium processors have 4 debug registers that can watch for reads/writes to the addresses stored in each register
- Can be configured to generate an exception when a write is made to one of the stored addresses
- By storing most recently used return addresses in the debug registers, writes to those registers can be prevented
- This strategy is more performant than the previously discussed virtual memory approach

Adaptive Defense Strategy

- Mem Guard implementation is significantly more expensive than using a canary word
  - Discussed in results section
- Adaptive strategy will use "canary mode" until an attack is detected at which point it will switch to Mem Guard
- Performance is only significantly degraded during attack but security remains high

Table of Contents

Attack Prevention Experiments

- Experimental penetrative tests were run on programs with and without StackGuard to gauge effectiveness
- Methodology involved simulating generic attacks on `roots`
- General success found with both Canary and MemGuard versions

Results

- Both highly effective
- Even for unprecedented, new attack styles
- Some effectiveness shown on similar attacks (namely `Perl`), but vulnerabilities when attacking things other than the function return address

Results

Vulnerable Program	Result Without StackGuard	Result With Canary StackGuard	Result With MemGuard StackGuard
`dip 3.3.7n`	`root` shell	program halts	program halts
`elm 2.4 PL25`	`root` shell	program halts	program halts
`Perl 5.003`	`root` shell	program halts irregularly	`root` shell
`Samba`	`root` shell	program halts	program halts
`SuperProbe`	`root` shell	program halts irregularly	program halts
`umount 2.5k/libc 5.3.12`	`root` shell	program halts	program halts
`wwwcount v2.3`	`httpd` shell	program halts	program halts
`zgv 2.7`	`root` shell	program halts	program halts

Figure: Results from experiments on various programs, from [1]

Overhead

- Of course, implementing added protection requires some overhead
- Experimental results were taken to find the ratio (%) of additional overhead relative to the non-StackGuard version of the program

Canary Overhead

- Pushing the canary onto the stack
- Checking the canary word is intact

Canary Overhead

Increment Method	Standard Run-Time	Canary Run-Time	% Overhead
`i++`	15.1	15.1	NA
`void inc()`	35.1	60.2	125%
`void inc(int *)`	47.7	70.2	69%
`int inc(int)`	40.1	60.2	80%

Figure: Canary overhead testing results

MemGuard Overhead

Significantly worse than canary

Increment Method	Standard Run-Time	MemGuard Register Run-Time	% Overhead	MemGuard VM Run-Time	% Overhead
`i++`	15.1	15.1	NA	NA	NA
`void inc()`	35.1	1808	8800%	34,900	174,300%
`void inc(int *)`	47.7	1820	5400%	40,420	123,800%
`int inc(int)`	40.1	1815	7000%	41,610	166,200%

Figure: MemGuard overhead testing results

Table of Contents

Overheads and Compromises

- It goes without saying that the MemGuard overhead is ridiculously bad
- In comparison, canary looks great, but alone, it is still almost twice as slow in cases
- As a result, we should strive to only use this version of StackGuard when absolutely needed

Overheads and Compromises

- Luckily, more recent developments in the area of buffer overflow prevention have been working on reconciling this

Table of Contents

Timeline of Protection in the GCC

1998: *StackGuard* is published
2001: IBM creates *ProPolice*, inspired by StackGuard
2005: A derivative of ProPolice is included in GCC 4.1

- `-fstack-protector` and `-fstack-protector-all`

2012: GCC 4.9 adds `-fstack-protector-strong`

These days, many Linux distributions (incl. Arch, Fedora, and Ubuntu) packages are compiled with this protection by default

StackGuard: a Pioneer?

- Indeed, StackGuard was the first of its kind
- Never actually implemented in GCC, but successors were
- This was a great first step, as it set the stage for buffer overflow attack prevention

References

[1] Crispan Cowan et al. "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks". In: *USENIX security symposium* 98 (1998), pp. 63–78.