Protecting Systems from Stack Smashing Attacks with StackGuard

Crispin Cowan

Steve Beattie, Ryan Finnin Day, Calton Pu,
Perry Wagle, and Erik Walthinsen

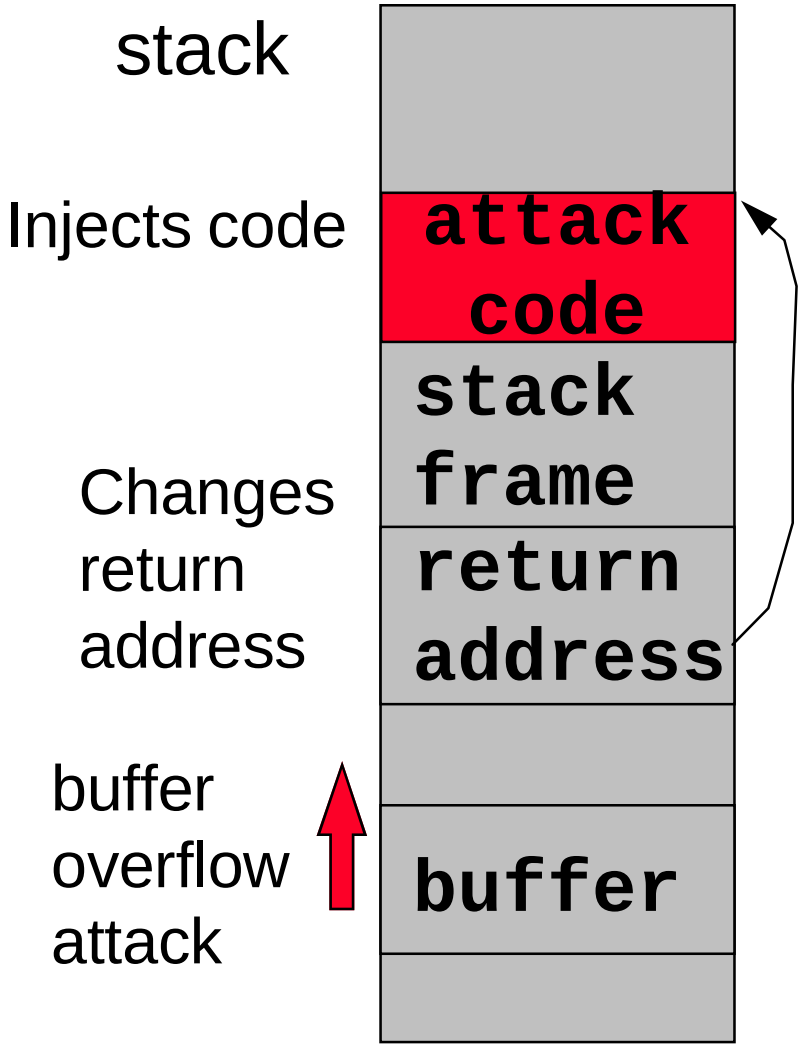
OREGON GRADUATE INSTITUTE
OF
SCIENCE & TECHNOLOGY

Executive Summary

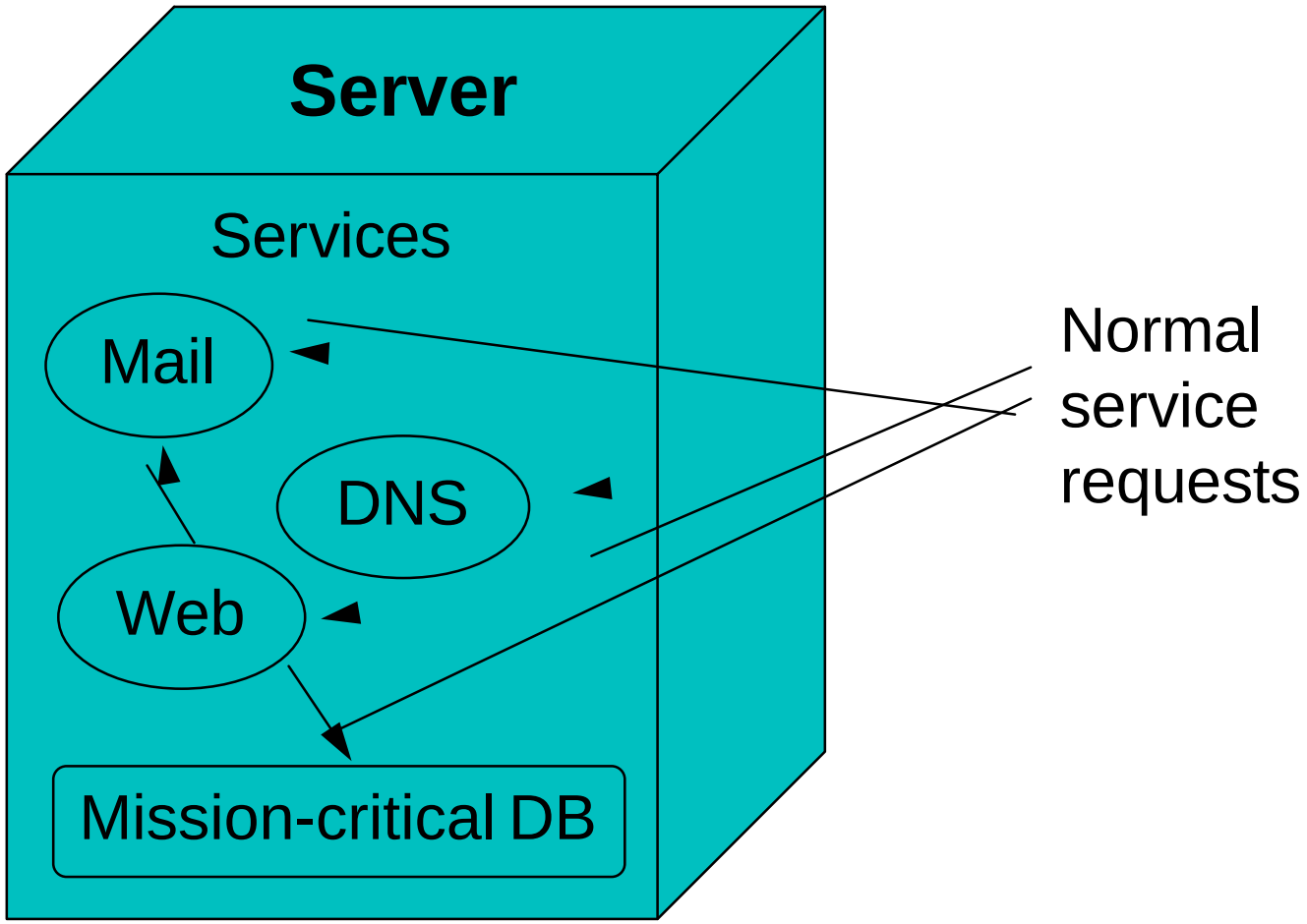
- Buffer overflow vulnerabilities are *still* rampant
  - » 8 of 13 CERT advisories *this year*
- StackGuard compiler:
  - » Protect any one program from buffer overflows
- To protect an entire system: Red Hat 5.1
  - » Must re-compile ***all*** programs
- So we did :-)
  - » How we did it
- Results: security, compatibility, performance

What's a Buffer Overflow Attack?

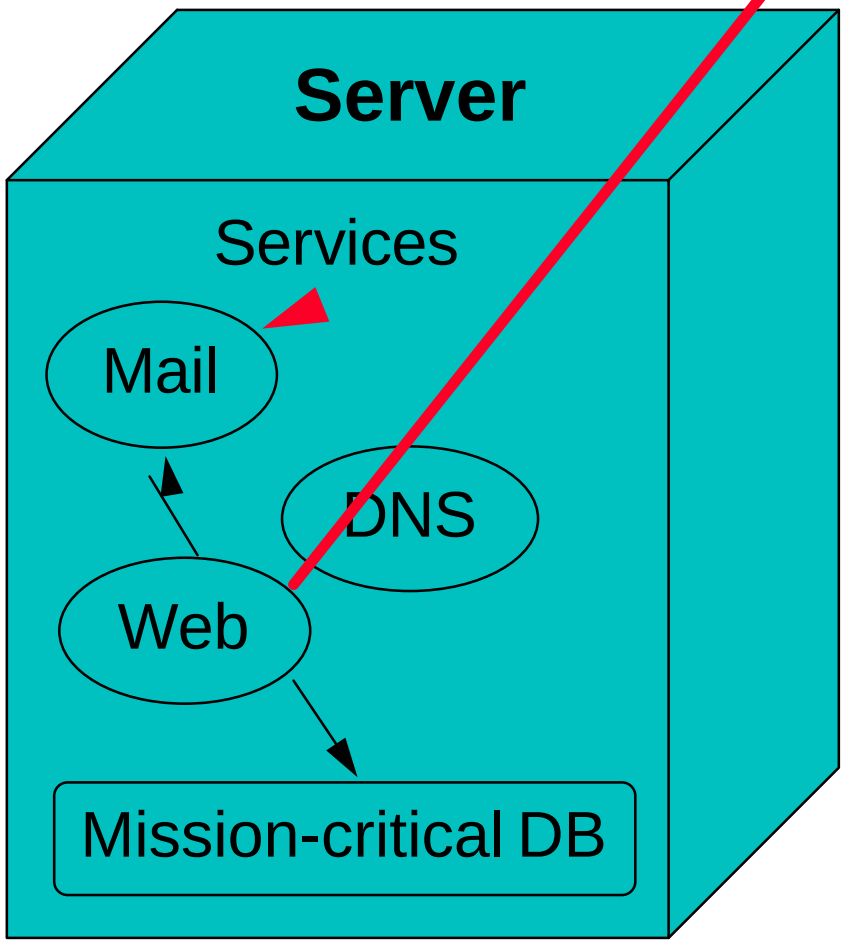
- Attacker feeds a big string to an input routine that does not do bounds checking
- String over-writes return address
- String injects code
- Function return jumps to injected code



Buffer Overflow Attacks



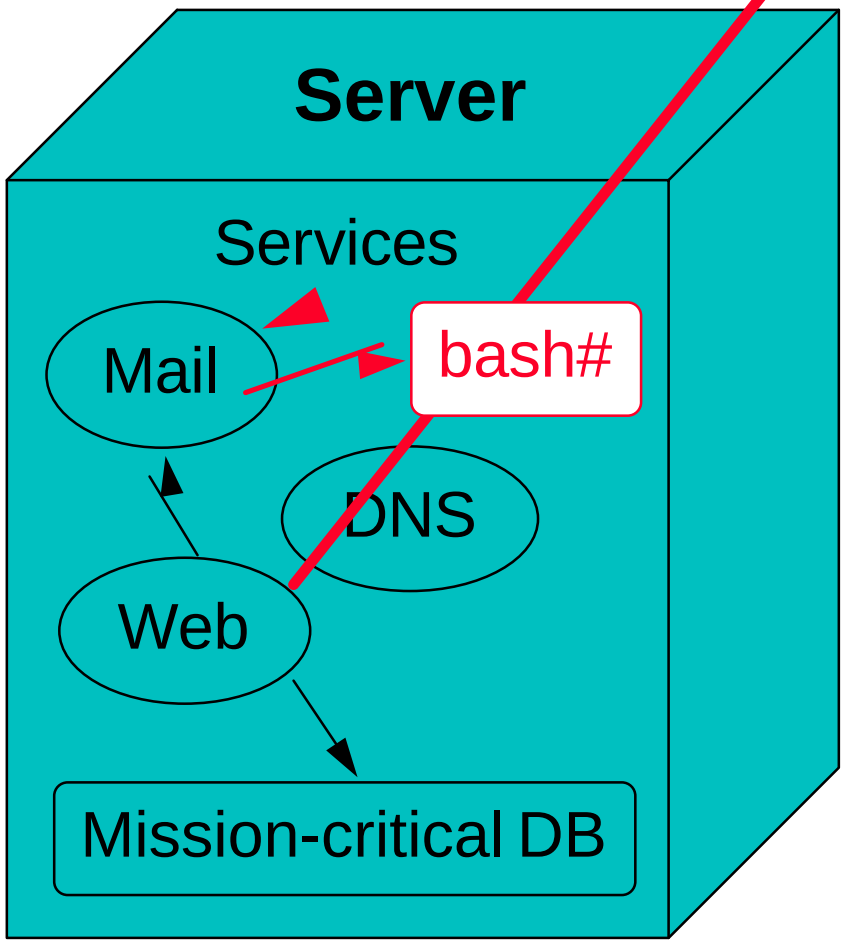
Example Application Attack



Attack

- Presents overflow string to root process

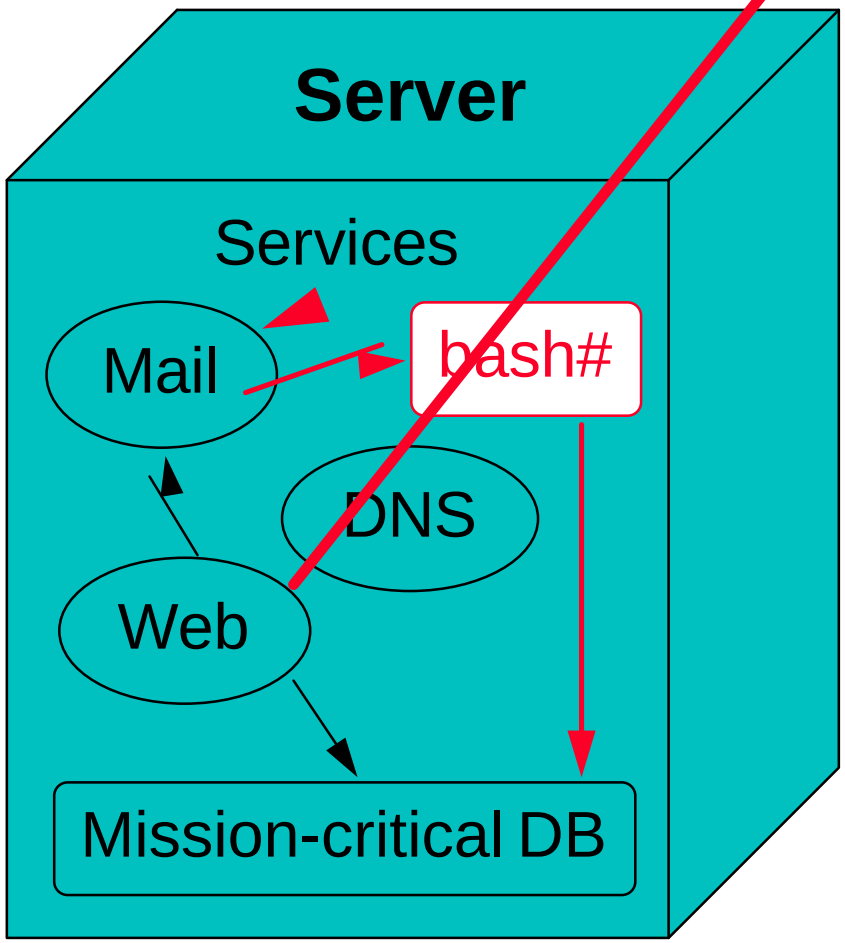
Example Application Attack



Attack

- Presents overflow string to root process
- Injected code exec's root shell

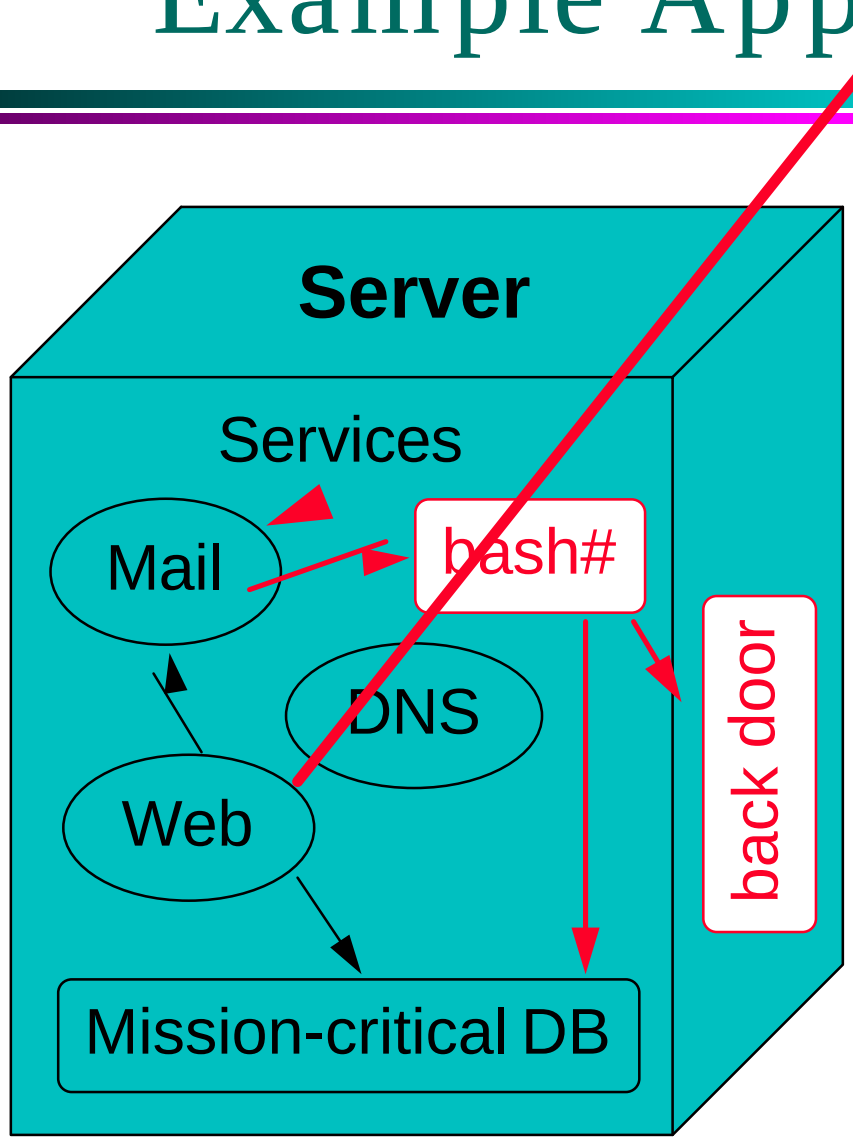
Example Application Attack



Attack

- Presents overflow string to root process
- Injected code exec's root shell
- Corrupt data

Example Application Attack

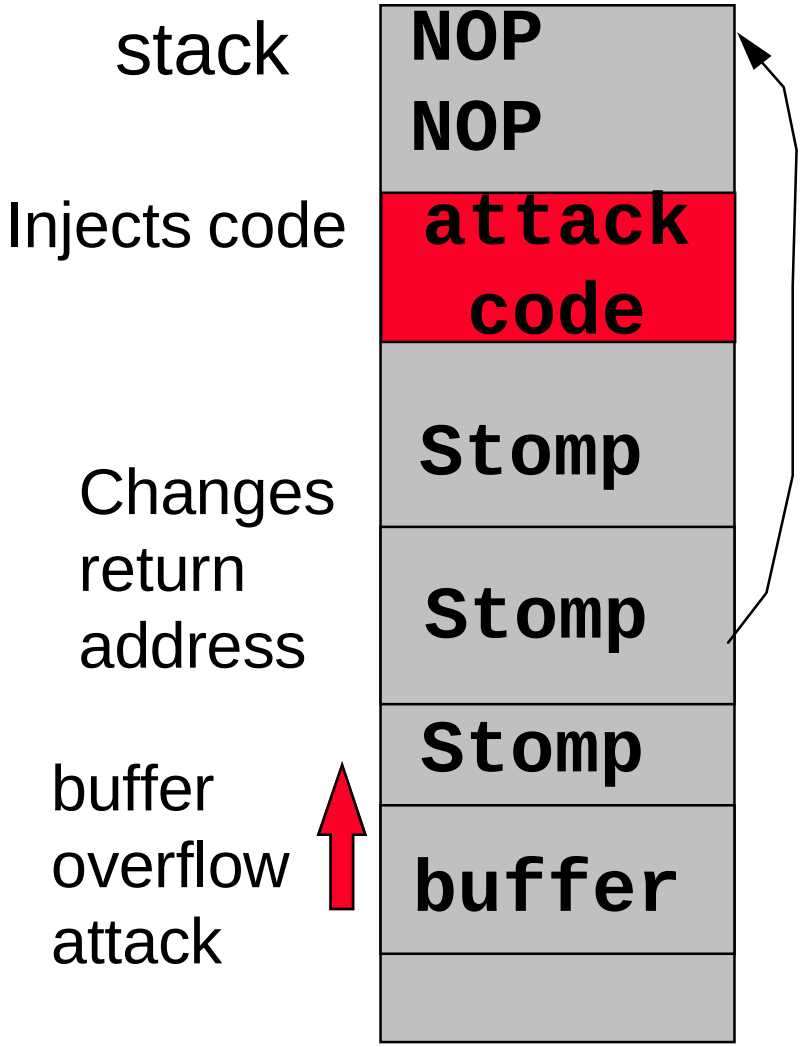


Attack

- Presents overflow string to root process
- Injected code exec's root shell
- Corrupt data
- Install rootshell back door

How Do Attackers *Create* Buffer Overflow Attacks?

- In principle, it's tricky
- In practice, there are cook-books
  - » Approximate location of return address
  - » Approximate start of attack code
- Would-be hacker need only find an unprotected buffer in trusted code
  - » Ample opportunities :-)

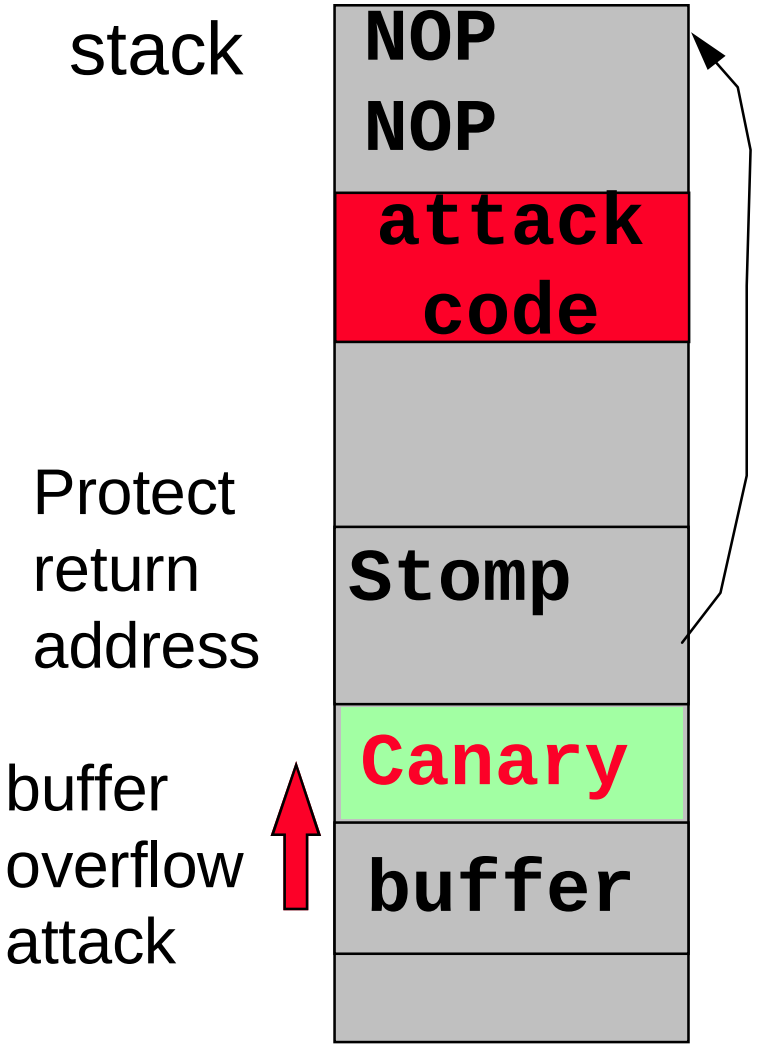


Buffer Overflows are *Still* Rampant

- Buffer overflows were a big deal in 1988: the Worm
- Still a big deal in 1999
  - » 8 of 13 CERT advisories *this year*
- No sign of abatement:
  - » Caused by C idiom of null-terminated strings mixed with static buffer size
  - » Only great care can eliminate them, e.g. OpenBSD security audit of *everything*
  - » Even auditing eventually rots: new patches introduce new bugs
  - » Auditing doesn't catch everything, e.g. **`lpd`**

StackGuard: Defeating Buffer Overflow Attacks

- Stack smash goes through
  - » Attack code injected
  - » Return address altered
- ***But*** Stack smash also smashes the **`Canary`**
  - » Function checks for **`Canary`** before returning
  - » If **`Canary`** smashed, program ***halts*** instead of yielding control to the attacker



Protecting the `Canary`

- Must prevent attacker from embedding canary in their attack
  - » Attacker includes canary in middle of attack string
  - » When program returns, canary is still there
- `Canary` value must be decidable
  - » Code has to check it on return
- Need a way to defend the canary
  - » Two defenses ...

Protecting the `Canary`: Random `Canary`

- Choose random canary value at exec() time
  - » Attacker can't learn random value from code inspection
  - » Program *can* deterministically validate canary value at return time
  - » Repeated guessing doesn't work, because it changes on each **`exec()`**
  - » Use a vector of canaries:
    - – One per function, mod 128

Protecting the **Canary**: "Terminator" **Canary**

- Canary is "null" (0), "CR", "LF" and EOF (-1)
  - » Most buffer overflows use C standard libraries
  - » Most library string functions stop on one of the above symbols
  - » Attacker can't embed a termination symbol in a string and expect the copy to proceed
  - » Protects the return address *beyond* the canary

Random vs. Terminator **Canary**

- Terminator:
  - » Faster: don’t have to look up canary value
- Random:
  - » More secure: there is an arcane attack that could theoretically defeat the terminator canary
- StackGuard supports both types of protection

Implementation: Modify **gcc** Function Code Generator

- **function_prolog**: Emit code to lay down a **canary** word
  - » Find right **canary** value for this function
  - » Push **canary** value onto stack

```
move func_num,r5
push canary_vector[r5]
```

- **function_epilog**: Emit code to verify a **canary** word
  - » Find right **canary** value for this function
  - » Compare with **canary** value on stack

```
move func_num,r4
move canary_vector[r4],r5
xor r4, top-of-stack
jnz canary_death_handler
add 4,stack_pointer
```

Protecting an Entire System: Red Hat Linux 5.1

- Need to re-compile every “vulnerable” program on your machine
- “Vulnerable” means:
  - » Program has more privilege than the attacker
  - » Program is running, or attacker can run program
  - » Attacker can provide input to the program
- But that’s hard to correctly determine
- Instead, we protected *everything*
  - » Evaluate computability, security, and performance
- 2 problems to solve
  - » Support shared libraries
  - » Stable build environment

Supporting Shared Libraries

- ELF shared libraries use PIC: Position Independent Code
- When in “PIC mode”
  - » Absolute references don’t work
  - » But you need a *per process* pointer to the random canary table
  - » Terminator canary solves the problem

Stable Build Environment

- Install *all* source and binary packages
- Build-order matters
  - » But source RPMs don't have explicit dependency information (yet)
- So we guess :-)
  - » Build everything
  - » Iterate until packages stop changing
- Rhed-Stone: time to recompile all of Red Hat
  - » 5+ hours on a dual PII-450 with fast-wide SCSI

Stable Build Environment: *Caution*

- Installing all this stuff enables lots of services
- You're vulnerable to attack while building
- Recommendation while building
  - » Firewall or disconnect from the network

Stable Build Environment: Some Packages Have Problems

- **`glibc-2.0.7-13`**
  - » Scanned assembly output for a keyword
    - – Word is in the comments emitted by StackGuard
  - » Looks for compiler switch support by compiling a null program and looking for error codes
    - – StackGuard has an added dependence on **__canary_death_handler()** so this test fails
- XFree86 moves a directory it does not own
  - » Repeated builds fail
  - » XFree86 comes late in the build :-(

Compatibility

- Near 100% compatibility
- Since last August:
  - » Running this laptop
  - » Running our group’s file server
  - » Several hundred downloads from our web site

Compatibility: Only Two Known Problems

- **`ld.so`**
  - StackGuarded **`ld.so`** breaks some binary-only applications that use **`libc5`**, e.g. Netscape, Star Office, Acroread, and WABI
  - Standard **`ld.so`** fixes the problem
- Kernel builds: StackGuard cannot build kernels
  - What would the kernel do if it detected a smash?
  - Solution: keep RPMs for standard and StackGuard versions of gcc
  - Switch compilers as necessary

Security: Penetration Experiments

- Collected exploits from **`Bugtraq`**, **`Linux-security`**, and **`comp.unix.security`**
  - » Vulnerable source, attack program
- Reproduce the attack with standard **`gcc`**
  - » Demonstrate getting **`root`** privilege
- Re-compile ***unmodified*** vulnerable program with StackGuard
  - » Demonstrate that attacked program ***halts***
- Of particular interest: Effectiveness in defending against ***future*** attacks
  - » Attacks released *after* our StackGuard release

Penetration Experiments

Vunlerable Program	Without StackGuard	With Canary StackGuard
samba	**root shell**	**program halts**
umount+libc	**root shell**	**program halts**
wwwcount 2.3	**httpd shell**	**program halts**
zgv 2.7	**root shell**	**program halts**

- **umount**
  - » Vulnerability is in **libc**, not the program itself
  - » Re-compiling **libc** with StackGuard is effective
  - » StackGuard **.o** files link with other files
- **samba, wwwcount**
  - » Vulnerabilities and exploits announced ***after*** StackGuard was built
  - » Demonstrates that StackGuard can defend against ***future*** attacks on ***unknown*** bugs

Penetration Experiments

Vunlerable Program	Without StackGuard	With Canary StackGuard
`dip 3.3.7n`	**`root` shell**	**program halts**
`elm 2.4 PL25`	**`root` shell**	**program halts**
`Perl 5.003`	**`root` shell**	**program halts irregularly**
`Superprobe`	**`root` shell**	**program halts irregularly**

- **`Superprobe`**
  - » Buffer is on the stack
  - » Overflow hits function pointer, not return address
  - » Overflow goes ***through*** the return address
  - » Only memory perturbation stops attack against canary
- **`Perl 5.003`**
  - » Buffer is not on the stack, it's in static data area
  - » Overflow hits **`longjmp`** buffer, not return address
  - » Only memory perturbation stops attack

New Penetration Experiments: Exploits Since the RH 5.1 Build

`xterm`:

- » Without StackGuard: root shell
- » In our Linux distribution: program halts

`lsof`:

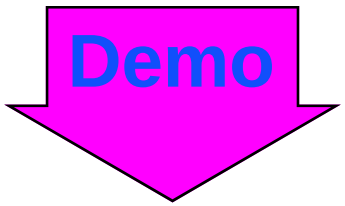
- » Without StackGuard: root shell

Note: `lsof` exploit works even *with* a non-executable stack kernel

- » In our Linux distribution: program halts

Penetration Test Demo

- Buffer overflow attack against **`dip`** program
  - » **`dip`** is **`setuid root`**, so it can dial the phone
  - » Attack quickly yields a root shell
- Re-compile ***unmodified*** source for **`dip`** with StackGuard compiler:
  - » Attack just causes **`dip`** to complain and die



Performance: StackGuard Overhead

- Privileged daemons: light compute load
  - » Only need to show that overhead is moderate
- Microbenchmarks:
  - » Identify the marginal increase in the cost of a function call
- Macrobenchmarks:
  - » Identify the overall increase in the time to execute a program protected by StackGuard

Microbenchmarks

- Cost is ***entirely*** in function call and return
- % Increase affected by base cost of function call
  - » Base: simple **`i++`** loop
  - » Side effect function call: **`void inc()`**
  - » Reference function call: **`void inc(int *)`**
  - » Applicative function call: **`int inc(int)`**
- Subtract base, compare StackGuard against standard compiler

Microbenchmarks

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%

Macrobenchmarks

- Real Time: overhead will change latency
  - » Few privileged programs are latency-sensitive
  - » Some are, e.g. high-bandwidth web server
- Compute Time: User+System Times
  - » **`root`** programs generally **are** the overhead
  - » Their compute time is the time the CPU is ***not*** spending on your application
  - » Thus compute time is the dominant concern

Macrobenchmarks: **`ctags`** and **`gcc`**

- **`ctags`**: cross-references C source code
  - » Light computation, lots of disk I/O
  - » Tests:
    - **`ctags linux/kernel/*.[ch]`** (37,000 lines)
    - **`ctags linux/*/*.[ch]`** (586,000 lines)
- **`gcc`**: GNU C Compiler
  - » Moderate computation, moderate disk I/O
  - » Test: **`make ctags`**

Macrobenchmark: `ctags`

Input	Version	User Time	System Time	Real Time
37,000 lines	Generic	0.41	0.14	0.55
	Canary	0.68	0.13	0.99
586,000 lines	Generic	7.74	2.08	10.2
	Canary	11.9	2.07	14.5

- Real Time Costs
  - » 42-80%
- Compute time: user+system
  - » 47-73%

Macrobenchmark: **gcc**

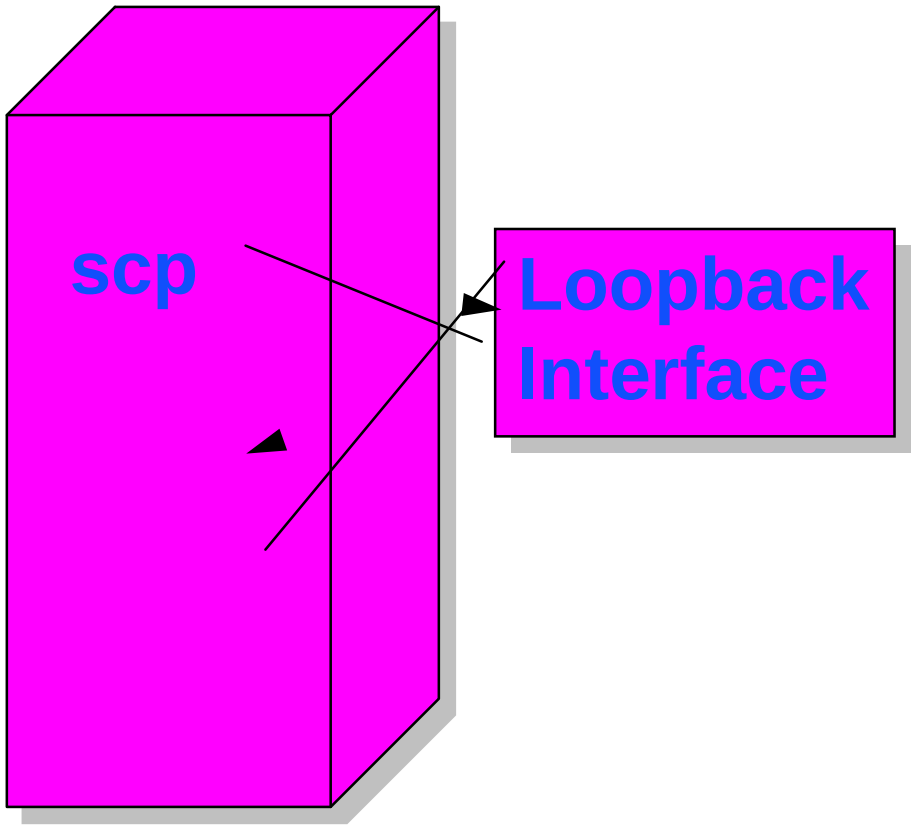
Version	User Time	System Time	Real Time
Generic	1.70	0.12	1.83
Canary	1.79	0.16	1.96

- Real Time Costs
  - » 7%
- Compute time: user+system
  - » 7%

Macrobenchmark: SSH

`scp bigsrc localhost:bigdst`

- Software encryption through the loopback interface
- No measurable performance difference between StackGuarded and standard SSH



Macrobenchmark: Apache

- Compare StackGuarded and unprotected Apache
- *Very* real application
  - » Performance sensitive
  - » Severe security exposure
- Measure performance with WebStone
  - » Worst-case: 8% penalty, average is a wash

StackGuard Protection	# of Clients	Connections per Second	Average Latency (ms)	Average Throughput (Mb/s)
No	2	34.44	58	5.63
No	16	43.53	358	6.46
No	30	47.2	603	6.46
Yes	2	34.92	57	5.53
Yes	16	53.57	295	6.44
Yes	30	50.89	561	6.48

Related Work

- Snarskii's FreeBSD Stack Integrity Check
  - » Similar integrity check to StackGuard's **`Canary`**
  - » Hand-crafted for **`libc`**
  - » Protects vulnerabilities in **`libc`**, but not in rest of program
- Solar Designer's Non-Executable Stack
  - » Useless to inject attack code onto the stack
  - » ***Can*** inject code into heap or static data
  - » Similar, but not identical protection to StackGuard
  - » **For added security, use both this and StackGuard**

“Halt?! But I *Need* That Daemon”

- Getting your daemons started again
- Reporting the intrusion

Re-Starting Daemons

- Simple case: daemons started by **`inetd`**
  - **`inetd`** starts daemon each time a request arrives
  - Can just let the daemon die, and **`inetd`** will start new ones when new requests arrive
- Harder case: persistent daemons, e.g. **`sendmail`**, **`inetd`**
  - Need a watch-dog; When watch-dog see daemon has died, re-starts it
  - Reduces security problem to simple fault-tolerance

Reporting Intrusion

- StackGuard produces intrusion alerts with ***very*** high confidence
  - » Should be reported to system IDS
  - » Shopping for IDS to report to
  - » Uses syslog by default, because it's everywhere

Stopping *Future* Attacks

- StackGuard is not a ***perfect*** solution
  - » There are buffer overflows that beat it
  - » It is better to patch a vulnerability
- ***But*** StackGuard provides protection for a broad selection of bugs
  - » Especially those you ***don't*** know about
  - » Gives you time to apply a patch when a vulnerability is announced

Conclusions

- StackGuard provides effective defense against most stack smashing attacks
  - » Stack smashing *still* the most common attack
  - » Can stop ***unknown*** stack smashing attacks
- Only requires re-compilation
- Compatible with existing technology
  - » Works with standard OS's and libraries
- Demonstrated effectiveness by building a defended system

Current Work

- Do a re-build of RH **5.2**
- Problem: glibc 2.0 is being a pain
  - » We put the **`canary_death_handler`** in **`libgcc`**
  - » **`glibc`** build barfs if **`canary_death_handler`** is present
- Solution in the works:
  - » Take **`canary_death_handler`** out of **`libgcc`** for purpose of building **`glibc`**
  - » Explicitly add **`canary_death_handler.o`** to each make line

Future Work

- Enhance StackGuard to protect non-stack data structures:
  - » Function pointers, **longjmp** buffers

Availability

- StackGuard compiler is GPL'd
- Associated library is LGPL'd
- Software on the web

`http://www.cse.ogi.edu/DISC/projects/immunix/StackGuard/`

Hacking

- If you want to test the robustness of StackGuard:
  - » Download it and put it on your machine
- If you ***really*** have to hack on a machine that's not yours, please use:
  - » **`gauntlet.cse.ogi.edu`**
- In either case, we'd love to hear about:
  - » Any attempted penetration of your site
  - » Any successful penetration against a StackGuarded machine