Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade

Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, Jonathan Walpole

Presented By: Chris Neasbitt

Outline

- Introduction
- Motivation
- Attack Anatomy
- Attack Defenses

Introduction

- Most common form of security vulnerability over the last ten years
- Most common vulnerability used for remote network penetration
- At least half of 1999 CERT advisories involve buffer overflows

Motivation

- Allow an attacker to do two necessary things
  - inject attack code
  - run attack code at elevated privilege levels
- Allows the attacker to attack the system remotely
- Easy to exploit

Attack Anatomy

- In order to exploit a buffer overflow the attacker must do two things
  - Arrange for suitable code to be available in the program's address space.
  - Get the program to jump to that code, with suitable parameters loaded into registers & memory

Attack Anatomy

- Placing code in the vulnerable program's address space
  - Inject It
    - supply a String containing native CPU instructions to the program
  - Make use of code available on the system
    - ex. if a program contains exec(arg) you might be able to change the pointer arg to point to “/bin/sh”

Attack Anatomy

- Causing the program to jump to the attack code
  - Activations Records
    - Overwrite a functions activation record in such a manner that cause the return pointer to point to the attack code
    - very prevalent
  - Function Pointers
    - overwrite a buffer close to a function pointer to cause the function pointer to point at the attack code
  - Longjmp Buffers
    - setjmp and longjmp are checkpoint/rollback functions
    - corrupt the state of the checkpoint buffer so longjmp calls the attack code

Attack Anatomy

- Combination techniques
  - Feed an overflowable automatic variable with a string that overwrites the return pointer and contains the executable code
    - simplest and most common attack
  - Also write the attack code to one buffer and overflow another to overwrite the return pointer
    - This is used when bounds checking exists but is incorrect

Attack Anatomy

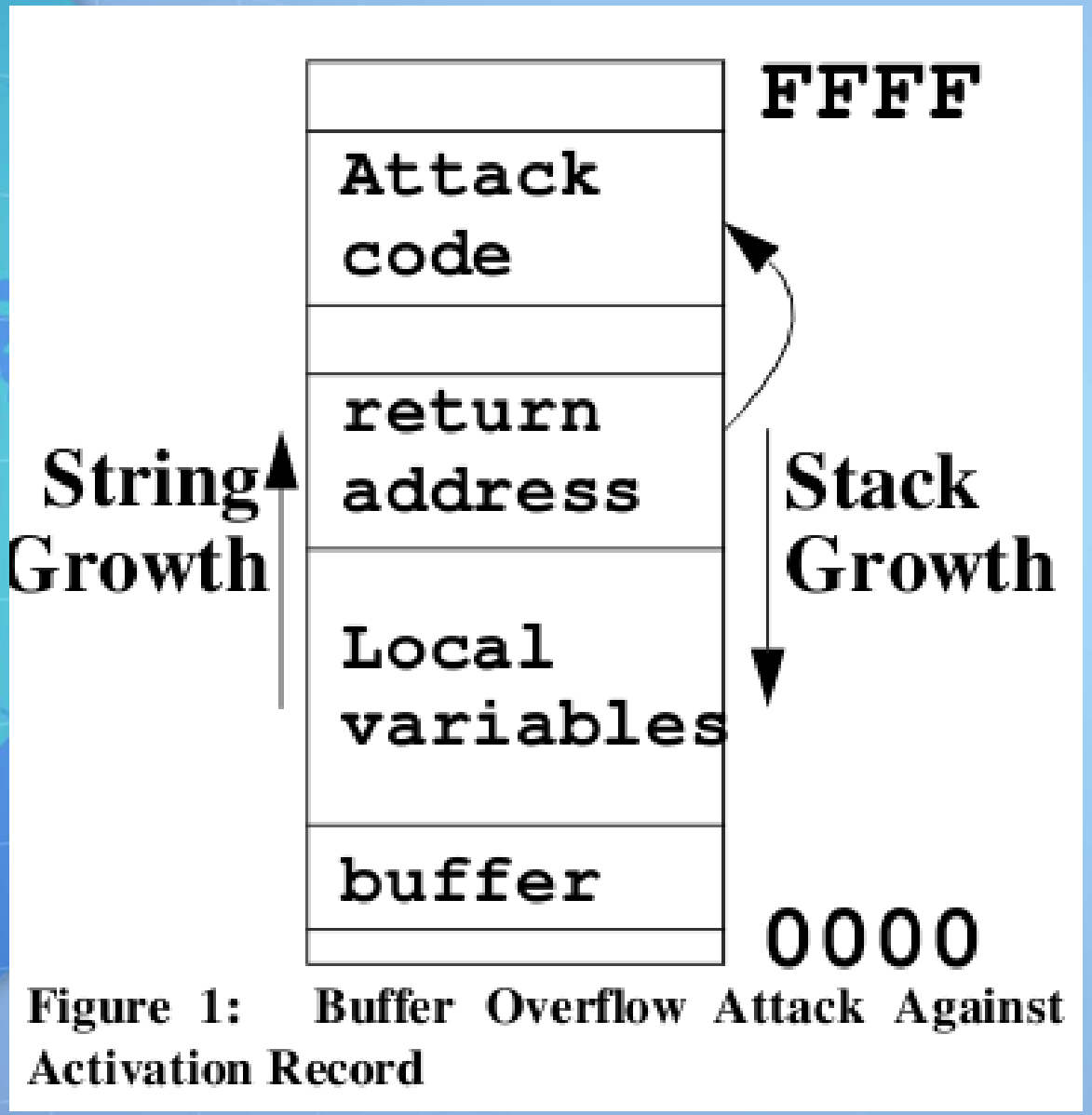

Figure 1: Buffer Overflow Attack Against Activation Record

Attack Defenses

- Programmer Oriented
  - writing correct code
- Operating Systems Oriented
  - make buffers non-executable
- Direct Compiler Approach
  - bounds check all array accesses
    - eliminates all buffer overflows but at high cost
- Indirect Compiler Approach
  - check integrity of all code pointers before dereferencing them
    - eliminates most buffer overflows at much lower cost

Attack Defenses

- Programmer Oriented
  - grep for vulnerable library calls like strcpy and sprintf
  - replace them with safer alternatives like strncpy and snprintf
  - code auditing teams
  - fault injection tools
    - helps search for vulnerable code
  -

Attack Defenses

- Non-Executable Buffers
  - make the data section of the code non-executable
  - this can be accomplished with the highest compatibility by making only the stack segment non-executable
    - virtually no legitimate programs need an executable stack
    - 2 exceptions
      - Signal Delivery
        - work around available in kernel patches
      - GCC Trampolines
        - not really used

Attack Defenses

- Array Bounds Checking
  - completely eliminates buffer overflows
  - implementations
    - Compaq C Compiler
      - checks all explict array references,
        - ex. a[1]
      - indirect references aren't checked
        - ex. *(a+3)
      - no bounds checking in subroutines
      - dangerous functions calls are still compiled without bounds checking

Attack Defenses

- Implementations Cont.
  - Jones & Kelly: Array Bounds Checking for C
    - gcc patch
    - derive a “base” pointer for each pointer expression and check pointer attributes to determine bounds
    - huge slowdown
      - ijk matrix multiplication, 30x slowdown
      - parts of SSH, 12x slowdown
      - some programs won't execute at all with this patch
        - elm

Attack Defenses

- Implementations Cont.
  - Purify: Memory Access Checking
    - uses “object code insertion” to instrument all memory access
    - uses a custom linker and library
    - not intended for production
    - 3 to 5 times slowdown
  - Type Safe Languages
    - Java
      - JVM is written is C so it could be vulnerable itself

Attack Defenses

- Code Pointer Integrity Checking
  - Tries to detect corrupted pointers before they are dereferenced
  - Does not solve all buffer overflows
  - better performance and compatibility than Array Bounds Checking
  - 3 implementations
    - Snarskii's custom libc for FreeBSD
    - StackGuard
    - PointGuard

Attack Defenses

- StackGuard
  - performs activation record integrity checking
  - written by the authors
  - implemented as a patch to gcc
  - places a “canary” value next to the return address in the activation frame
  - before activation record is removed from the stack the canary value is checked to see if it was overwritten
    - overwritten canary values means that a buffer has overflowed

Attack Defenses

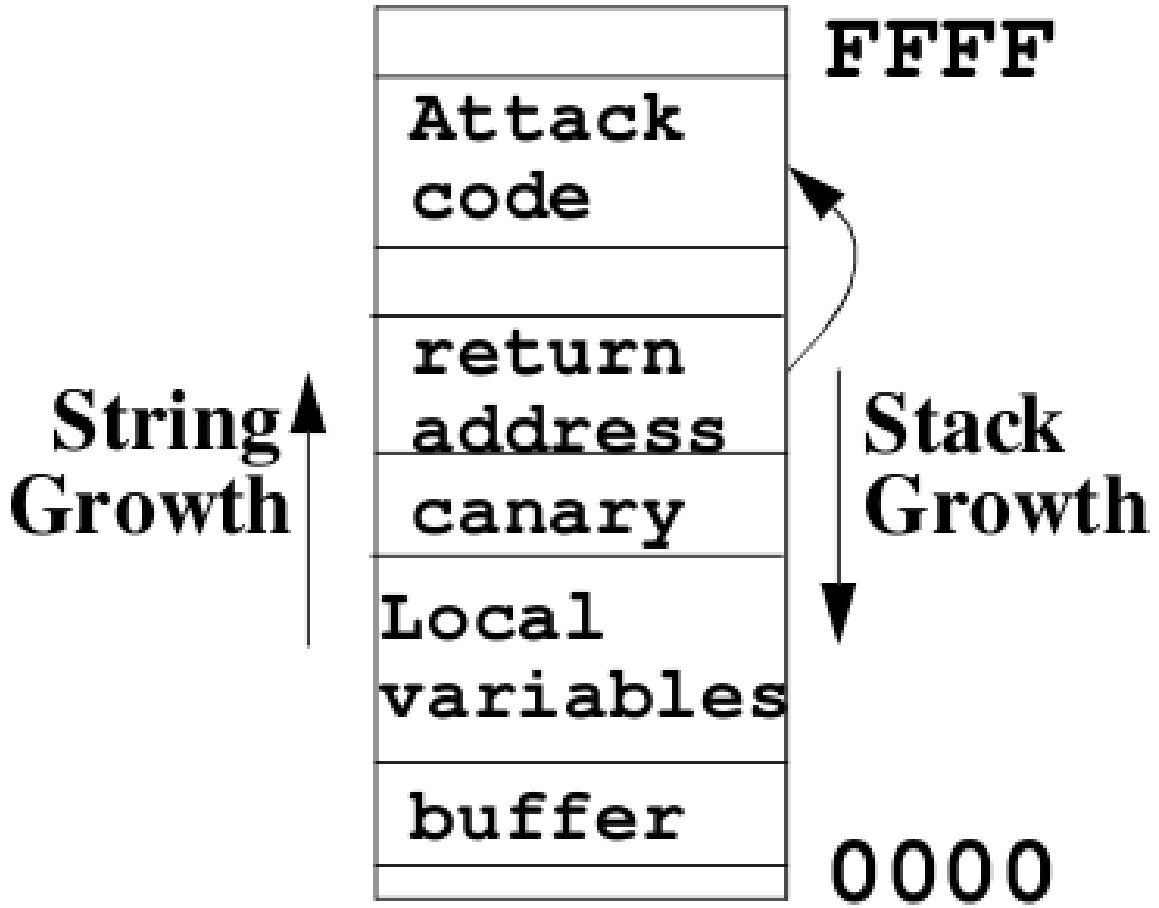

Figure 2: StackGuard Defense Against Stack Smashing Attack

Attack Defenses

- Canary Forgery Prevention
  - Terminator Canary
    - Fill the canary with terminator symbols
      - ex. (null), CR, LF, EOF
    - Attacker cannot embed these symbols into the overflow string because C lib string functions will terminate on encountering them
  - Random Canary
    - Generate a 32-bit random number
    - never disclosed
    - new one is generated for each program execution

Attack Defenses

- StackGuard Security
  - derived from the notion of quasi-invariants to assure the correctness of incremental specializations
    - quasi-invariants
      - something that changes but only occasionally
    - specialization
      - deliberate change to a program that is correct only under certain conditions
    - StackGuard's quasi-invariant is the fact that an active function's return pointer should not change
    - an attacker's attempt to overwrite the return pointer would be considered invalid as it violates the quasi-invariant

Attack Defenses

Table 1: StackGuard Penetration Resistance

Vulnerable Program	Result Without StackGuard	Result with 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
`Samba`	**`root`** shell	program halts
`SuperProbe`	**`root`** shell	program halts irregularly
`umount 2.5K/libc 5.3.12`	**`root`** shell	program halts
`wwwcount v2.3`	**`httpd`** shell	program halts
`zgv 2.7`	**`root`** shell	program halts

Attack Defenses

- StackGuard Performance
  - tested with WebStone benchmark

Table 2: Apache Web Server Performance With and Without StackGuard Protection

StackGuard Protection	# of Clients	Connections per Second	Average Latency in Seconds	Average Throughput in MBits/Second
No	2	34.44	0.0578	5.63
No	16	43.53	0.3583	6.46
No	30	47.2	0.6030	6.46
Yes	2	34.92	0.0570	5.53
Yes	16	53.57	0.2949	6.44
Yes	30	50.89	0.5612	6.48

Attack Defenses

- PointGuard
  - performs code pointer integrity checking
  - generalization of StackGuard
  - places canaries next to all code pointers
  - still in development at time of writing
  - 2 main development issues involved
    - canary allocation
    - canary checking

Attack Defenses

- CPIC Compatibility and Performance
  - code pointers are deferenced far less frequently in the vast majority of programs than arrays accessed
  - arrays have no innate bounds attributes so they must be inferred
  - maintaining “sizeof(int) == sizeof(void *)” allows no extra information to be stored with the array itself
  - violating “sizeof(int) == sizeof(void *)” destroys code compatibility

Attack Defenses

- Combination Defenses
  - since no effective bounds checking compiler exists combinations of defenses could be used

Table 3: Buffer Overflow Attacks and Defenses

		Attack Code Location			
		Resident	**Stack Buffer**	**Heap Buffer**	**Static Buffer**
Code Pointer types	**Activation Record**	StackGuard	StackGuard, Non-executable stack	StackGuard	StackGuard
	Function Pointer	PointGuard	PointGuard, Non-executable stack	PointGuard	PointGuard
	Longjmp Buffer	PointGuard	PointGuard, Non-executable stack	PointGuard	PointGuard
	Other Variables	Manual PointGuard	Manual Point-Guard, Non-executable stack	Manual Point-Guard	Manual Point-Guard

Questions?