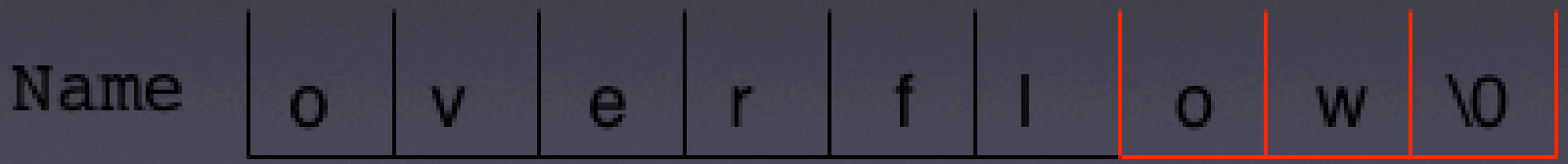
Buffer Overflows

Christian Klein <kleinc@cs.bonn.edu>

What is a buffer overflow?

- filling a *buffer* beyond its bounds
- typically a **char[]** (“c-string”)



char[]

- C datatypes: numbers, pointers, vectors, structs, but no character strings
- Length information is inbound

stack based buffer overflows

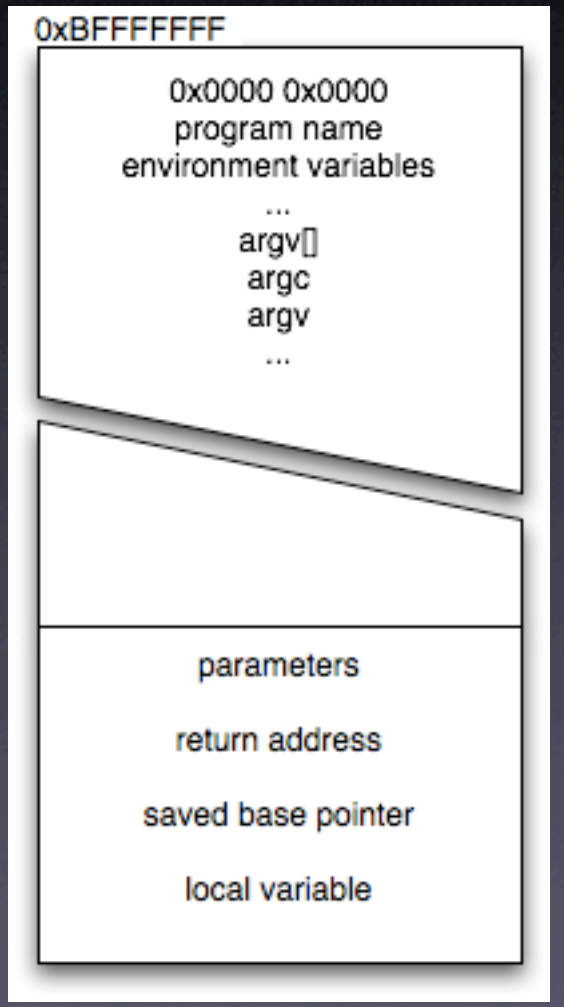
- known (at least) since 1988 (*Morris Worm*)
- most common security vulnerability (more than 1000 hits on Bugtraq)
- hard to automatically spot, easy to exploit
- exploits the fact that the stack is executable

stack 101

- what is the stack?
  - the memory area where *automatic variables* are stored
  - a LIFO structure with pop and push operations
  - grows from 0xBFFFFFFF down
    - (remind: *sub* enlarges the stack, *add* shortens it)

registers

- the stack pointer (*esp*) points to the top of stack, (TOS)
- the base pointer (*ebp*) points to the top of the current *stack frame*
- the instruction pointer (*eip*) points to the next machine instruction



stackframe allocation

- *function prologue*
  - the *eip* is saved on the stack (*call*)
  - the *ebp* is saved on the stack (*push %ebp*)
  - the new frame is created (*mov l%esp, %ebp*)
  - stack space is allocated (*subl $0x0c, %esp*)

stackframes deallocation

- *function epilogue*:
  - stack space is deallocated (*addl $0x0c, %esp*)
  - the stack frame is deleted (*movl %ebp, %esp*)
  - the saved base pointer is loaded (*pop %ebp*)
  - the saved instruction pointer is loaded and program flow continues (*ret*)

stack overflow, example 1

- what's happening in memory?
  - the buffer is allocated to hold 8 bytes
  - the next 4 bytes are the long int
  - the next 4 bytes are the saved ebp
  - the next 4 bytes are the saved eip
  - so, our strcpy() overwrites the saved eip!

stack overflow, example 2

- what's happening in memory?
  - in *myfunc*, a variable is declared and initialized with the address of itself - two word sizes (8 bytes)
  - that word is incremented by 10
  - that word was the saved instruction pointer and we just skipped a instruction

conclusion
so far

- saved base pointer and saved instruction pointer are overwritten
- instruction pointer is filled with information from stack
- we can change the control flow
- can we do something useful with that? YES!

shellcode
quick and dirty

- it's machine code that is injected into the memory
- platform dependent
- "a science on its own"
  - compact size
  - zero byte free
- available on the internet ;-)

shellcode
an example

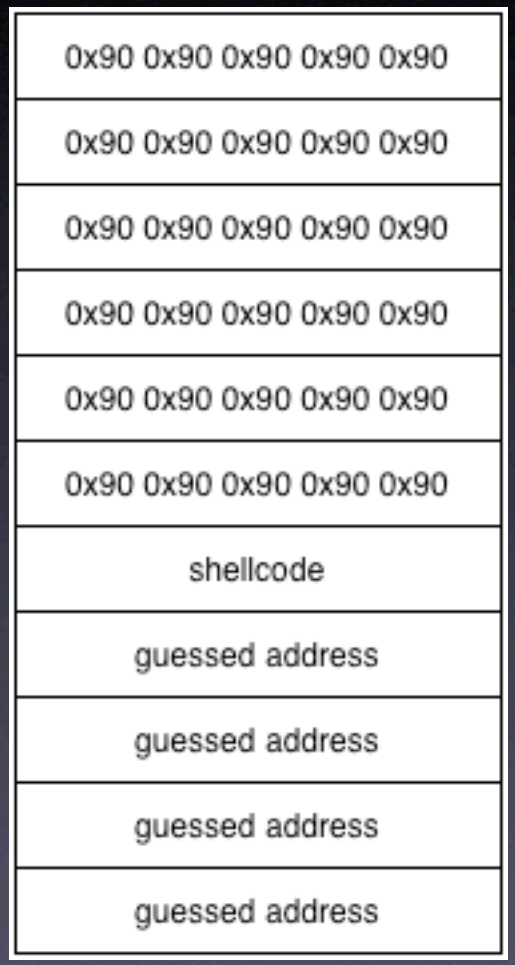
```
char shellcode[]=
    "\x31\xc0"      /* xorl  %eax,%eax   */
    "\x50"          /* pushl %eax        */
    "\x68""//sh"    /* pushl $0x68732f2f */
    "\x68""/bin"    /* pushl $0x6e69622f */
    "\x89\xe3"      /* movl  %esp,%ebx   */
    "\x50"          /* pushl %eax        */
    "\x53"          /* pushl %ebx        */
    "\x89\xe1"      /* movl  %esp,%ecx   */
    "\x99"          /* cdql              */
    "\xb0\x0b"      /* movb  $0x0b,%al   */
    "\xcd\x80"      /* int   $0x80       */
```

stack overflow
the exploit

- todo:
  - insert shell code (easy)
  - set return address to the address of the shellcode (tricky)
  - let the process jump into shellcode (just sit down and watch)

how to find the address of the shellcode?

- described by **Aleph One** in "*Smashing the stack for fun and profit*"
- helps us to guess: *__asm__("movl %esp, %eax")*
- *nop-sled* for not-so-acurate guessing
- works for local and remote exploits

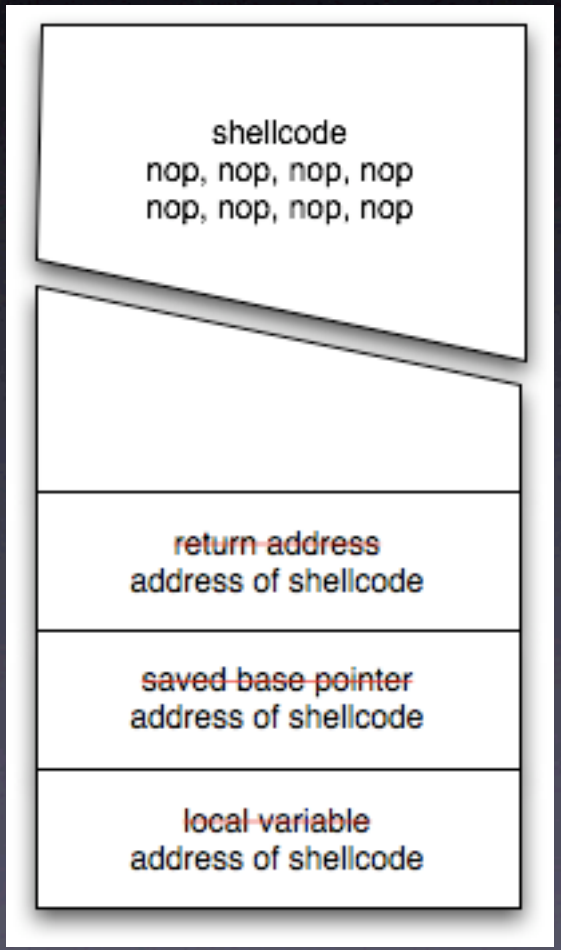


how to calculate the address of the shellcode

- little trick: put shellcode in environment variable
- advantage:
  - fixed address
  - works with tiny buffers
- address = 0xbfffffff - (4 + strlen(argv[0]) + 1 + strlen(envp[n]))
- disadvantage: works only local

stack overflow example 3

- the first exploit



the usual suspects

- all string manipulating functions
  - gets, strcpy, strcat, sprintf
  - always use the safer version: fgets, strncpy, strncat, snprintf
- memcpy with unchecked length

your task

- review source code.
- also / especially operating system code.

heap based overflows

- there is not a standard way
- different approaches
- less in focus of security software (StackGuard, protect_stack)

heap based overflows

- the heap is an area of memory that is dynamically allocated by the application.
  - the data section is initialized at compile time
  - the bss section is initialized at run time (zero filled)
  - also heap is RWX on most architectures

heap based overflows

- the heap grows up from a low address
- memory is usually (historically?) allocated with the brk() system call, which readjusts the *end_data_segment* variable

heap based overflows

- not as “standard” as stack based overflows
- usually no direct influence of the code flow
- might be even harder to detect

heap based overflows

- example 1:

heap based overflows

- variables are allocated on the heap
- *filename* is overwritten by *comment*
- we can append a single line of code to an arbitrary file by controlling the filename
- other possibilities: authentication state, permissions, shell scripts (startup scripts), ...

heap based overflows

- example 2:

heap based overflows

- like other variables, function pointers can be overwritten
- it's also possible to call shellcode:
  - place shellcode in environment
  - overwrite function pointer with address of shellcode

endangered data

- data on heap is usually more sensitive:
  - static buffers of libc functions
  - FILE structures, DIR structures
  - exit handlers
  - meta data of malloc

lab session

- write your own exploit
- master gera's challenges: http://community.core-sdi.com/~gera/InsecureProgramming/
- create documentation for everything

links

- http://untergrund.bewaff.net/~chris/bo/
- http://www.enderunix.org/docs/eng/bof-eng.txt
- http://www.insecure.org/stf/smashstack.txt
- http://www.w00w00.org/files/articles/heaptut.txt
- http://community.core-sdi.com/~gera/InsecureProgramming/