

## Operating System Security

### Trojan Horses

- Does NOT self-replicate
- Free program made available to unsuspecting user
  - Actually contains code to do harm
- Place altered version of utility program on victim's computer
  - trick user into running that program
  - la
  - /usr/mal/ls
- Rootkits
- Remote Access Tools
  - PCAnywhere
  - Laplink
  - Back Orifice

COMP 6370 – Buffer Overflows – Lecture 8

## Login Spoofing

Login:

Login:

(a)
(b)

(a) Correct login screen  
(b) Phony login screen

COMP 6370 – Buffer Overflows – Lecture 8

## Worms

- A viral or reproductive program that copies and spreads itself without associating with a particular host program.
- Worms date back to the Morris Worm.
- Classifying worms (Nachenberg 99)
  - By transport mechanism
    - Email Worms
    - Arbitrary Protocol Worms spread via protocols other than email protocols such as TCP/IP sockets
  - By launching mechanism
    - Self-launching worms ex. Morris Worm (rare)
      - Script viruses such as bubbleboy that exploit unpatched Outlook qualify
    - User-launched worms must be executed by a user and therefore require a degree of social engineering
    - Hybrid launch use both mechanisms.

COMP 6370 – Buffer Overflows – Lecture 8

## Logic Bombs

- Company programmer writes program
  - potential to do harm
  - OK as long as he/she enters password daily
  - ff programmer fired, no password and bomb explodes

COMP 6370 – Buffer Overflows – Lecture 8

## Trap Doors

```

while (TRUE) {
  printf("login: ");
  get_string(name);
  disable_echoing();
  printf("password: ");
  get_string(password);
  enable_echoing();
  v = check_validity(name, password);
  if (v) break;
}
execute_shell(name);

```

```

while (TRUE) {
  printf("login: ");
  get_string(name);
  disable_echoing();
  printf("password: ");
  get_string(password);
  enable_echoing();
  v = check_validity(name, password);
  if (v || strcmp(name, "zzzzz") == 0) break;
}
execute_shell(name);

```

(a)
(b)

(a) Normal code.  
(b) Code with a trapdoor inserted

COMP 6370 – Buffer Overflows – Lecture 8

## Buffer Overflow

(a)

(b)

(c)

- (a) Situation when main program is running
- (b) After program A called
- (c) Buffer overflow shown in gray

COMP 6370 – Buffer Overflows – Lecture 8

## Generic Security Attacks

### Typical attacks

- Request memory, disk space, tapes and just read
- Try illegal system calls
- Start a login and hit DEL, RUBOUT, or BREAK
- Try modifying complex OS structures
- Try to do specified DO NOTs
- Convince a system programmer to add a trap door
- Beg admin's sec'y to help a poor user who forgot password



COMP 6370 – Buffer Overflows – Lecture 8



7

## Design Principles for Security

1. System design should be public
2. Default should be no access
3. Check for current authority
4. Give each process least privilege possible
5. Protection mechanism should be
  - simple
  - uniform
  - in lowest layers of system
6. Scheme should be psychologically acceptable

And ... keep it simple



COMP 6370 – Buffer Overflows – Lecture 8



8

## RC5-64

- On 14-Jul-2002, a relatively characterless PIII-450 in Tokyo returned the winning key to the distributed.net keyserver.
  - The key 0x63DE7DC154F4D039 produces the plaintext output:
  - The unknown message is: some things are better left unread
  - So, after 1,757 days and 58,747,597,657 work units tested the winning key was found!
- While it's debatable that the duration of this project does much to devalue the security of a 64-bit RC5 key by much, we can say with confidence that RC5-64 is not an appropriate algorithm to use for data that will still be sensitive in more than several years' time.
  - The next time someone bemoans the public's short attention span or need for instant gratification you should remind them what 331,252 people were able to accomplish by joining together and working for nearly five years. distributed.net's RC5-64 project clearly shows that even the most ambitious projects can be completed by volunteers thanks to the combined power of the internet and distributed computing.



COMP 6370 – Buffer Overflows – Lecture 8



9

## RC5 Stats

- Ignoring artificially high numbers resulting from network difficulties, we completed 86,950,894 workunits on our best day.
  - This is 0.12% of the total keyspace meaning that at our peak rate we could expect to exhaust the keyspace in 790 days.
- Our peak rate of 270,147,024 kkeys/sec is equivalent to 32,504 800MHz Apple PowerBook G4 laptops or 45,998 2GHz AMD Athlon XP machines or (to use some rc5-56 numbers) nearly a half million Pentium Pro 200s.
- Over the course of the RC5-64 project, 331,252 individuals participated. We tested 15,769,938,165,961,326,592 keys.
- Sources: Charles Iser
  - <http://www.distributed.net/pressroom/news-20020926.html>
  - [http://www.rsasecurity.com/news/releases/pr.asp?doc\\_id=1400](http://www.rsasecurity.com/news/releases/pr.asp?doc_id=1400)



COMP 6370 – Buffer Overflows – Lecture 8



10

## Review: Preparing a Program for Execution

- Use an editor to enter the source code into main memory and save it on a disk as a source file.
- Use a compiler program to translate the source program into machine languages.
  - In Ada, you may need to make corrections to your code to get the code to compile, C will usually compile the first time and the errors become apparent at run-time.
- When the source program is error-free, the compiler saves its machine-language translation as an object file.
- Call the linker or binder to combine your object program with additional object files needed for your program to execute.
  - Generally, the linker saves the final result as an executable program on disk.



COMP 6370 – Buffer Overflows – Lecture 8



11

## Useful Definitions

- The **heap** is an area of memory reserved for data that is created at runtime -- that is, when the program actually **executes**.
  - malloc() or new
- The **stack** is an area of memory used for data whose size can be determined when the program is **compiled**.
- When contiguous chunks of the same data type are allocated, the memory region is known as a **buffer**.
- **Buffer overflow** occurs by writing past the end of an array.
- **Bounds checking** refers to programmer and compiler strategies for preventing buffer overflows.



COMP 6370 – Buffer Overflows – Lecture 8



12

## C, an average programming language

- C is inherently unsafe – programs may overflow buffers at will.
- No runtime checks that prevent writing past the end of a buffer.
- Reading or writing past the end of a buffer can cause a number of diverse behaviors
  - Programs may act in strange ways
  - Programs may fail completely
  - Programs may proceed without any noticeable difference in execution.



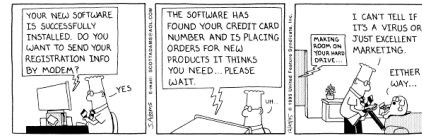
COMP 6370 – Buffer Overflows – Lecture 8



13

## Buffer Overrun Side Effects

- Depend on:
  - How much data are written past the buffer bounds
  - What data (if any) are overwritten when the buffer gets full and spills over
  - Whether the program attempts to read data that are overwritten during the overflow



Copyright © 1992 United Feature Syndicate, Inc. Redistribution in whole or in part prohibited.



COMP 6370 – Buffer Overflows – Lecture 8



14

## Security versus Poor Programming

- Consider a boolean flag in main memory
  - The flag determines whether the user running the program can access private files
  - The overflow overwrites the boolean flag
  - Illegal access to the files provided to attacker
- Stack Smashing
  - Careless use of data buffers allocated on a program's runtime stack (i.e. local variables and function arguments)
  - Attacker can usually run arbitrary code
    - Place attack code somewhere (i.e. code to invoke a shell)
    - Overwrite the stack in such a way that control gets passed to the attack code



COMP 6370 – Buffer Overflows – Lecture 8



15

## Heap Overflow versus Stack Overflow

- Common goal of overflow attacks are root shells
- Attacks are typically against a particular architecture (OS/machine combination)
- One common technique is to find a buffer overflow in a suid program
  - ex. lpr, xterm and eject to name a few
- Heaps are harder to exploit because they are dynamic, not static.
  - programming strategy is to new or malloc() everything
  - main protection is that fewer people know how to exploit heap overflows
  - Generally takes longer to set up a heap overflow attack



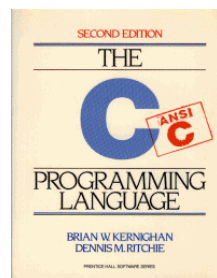
COMP 6370 – Buffer Overflows – Lecture 8



16

## Weak C Functions

- strcpy()
- strcat()
- sprintf()
- scanf()
- sscanf()
- fscanf()
- vfscanf()
- vscanf()
- vsscanf()
- streadd()
- strcpy()
- strtrns()



COMP 6370 – Buffer Overflows – Lecture 8



17

## strcpy versus strncpy

- char \*strcpy(s, ct)
  - copy string ct to string s, including '\0'; return s
- char \*strncpy(s, ct, n)
  - copy at most n characters of string ct to s; return s
  - pad with '\0's if t has fewer than n characters
- explicit check

```
if(strlen(src) >= dst_size) {  
    /* raise an error condition */  
}  
else {  
    strcpy(dst, src);  
}
```
- alternate

```
strncpy(dst, src, dst_size - 1);  
dst[dst_size - 1] = '\0';
```



COMP 6370 – Buffer Overflows – Lecture 8



18

## realpath()


- `realpath(3C)` C Library Functions `realpath(3C)`

**NAME**  
`realpath` - returns the real file name


**SYNOPSIS**  

```
#include <stdlib.h>
#include <sys/param.h>
char *realpath(char *file_name, char *resolved_name);
```

**DESCRIPTION**  
`realpath()` resolves all links and references to `..` and `...` in `file_name` and stores it in `resolved_name`.  
 It can handle both relative and absolute path names.  
 For absolute path names and the relative names whose resolved name cannot be expressed relatively (for example, `../rel-dir`), it returns the resolved absolute name.  
 For the other relative path names, it returns the resolved relative name.  
`resolved_name` must be big enough (MAXPATHLEN) to contain the fully resolved path name.



COMP 6370 – Buffer Overflows – Lecture 8




## Bounds Checking: A Good Thing


- create your own security problem

```
char buf[1024];
int i = 0;
char ch;
while ((ch = getchar()) != '\n') {
    if (ch == -1) break;
    buf[i++] = ch;
}
```

- Almost any C function that can read in a character is a candidate for an overflow.




COMP 6370 – Buffer Overflows – Lecture 8




## Smashing Stacks and Heaps

- **Regions of memory that are usually present:**
  - program arguments and the program environment
  - the program stack (static)
    - stack usually grows as the program executes
    - grows toward the heap
  - the heap (dynamic)
    - normally grows toward the stack
  - Block Storage Segment (BSS) contains globally available data (global variables)
    - BSS segment normally zeroed out at start-up
  - The data segment contains initialized globally available data such as global variables
    - initialized at declaration time
  - Text segment contains the read-only program code




COMP 6370 – Buffer Overflows – Lecture 8




## Stack Overflow Attack Outline

1. Find a stack-allocated buffer we can overflow that allows us to overwrite the return address of the stack frame
2. Place some hostile code in memory to which we can jump when the function we're attacking returns
3. Write over the return address on the stack with a value that causes the program to jump to our hostile code




COMP 6370 – Buffer Overflows – Lecture 8




## UNIX Exploit Example

- **Objective: Get a shell**
  - Long-term: escalate shell privilege to "root"
- **Compile attack code**
  - extract the binary and insert it into the buffer we are overrunning
  - insert the code snippet before or after the return address over which we have to write depending on space limitations
- **Figure out exactly where the overflow code should jump to.**
  - place that address at the exact proper location in the buffer in such a way that it overwrites the normal return address




COMP 6370 – Buffer Overflows – Lecture 8




## Exploit Outline

Position	Contents
Start of Buffer	Our exploit code might fit here or
End of Buffer	Our exploit code might fit here or
Other Vars	Our exploit code might fit here or
Return Address	A jump-to location that will cause our exploit to run
Parameters	Our exploit code if it did not fit elsewhere
Rest of Stack	Our exploit code, continued, and any data our code needs



COMP 6370 – Buffer Overflows – Lecture 8



## Haiku from Dildog, COTDC

Throughout these ages  
our operating systems  
infested by bugs

The ignorant world  
turns to Windows for safety  
Safety from themselves

It is now the time  
for the world to realize  
that we all feel pain



COMP 6370 – Buffer Overflows – Lecture 8

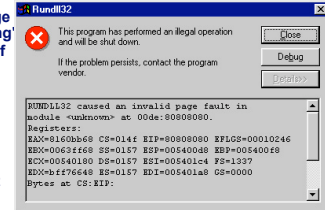


25

## Notes from the Cult of the Dead Cow

- To get this to happen, I fed a string of 0x80 bytes into a popular conference package called 'Microsoft Netmeeting' through the address field of a 'speeddial' shortcut.

- EIP happens to be 0x80808080.
  - Guess what?
  - That's good!
  - I found a stack overflow!
- Now all I have to do is craft my exploit string to have some fun code inside, and tweak four of those 0x80 bytes to point to my exploit string.



[http://www.cultdeadcow.com/cDc\\_files/cDc-351](http://www.cultdeadcow.com/cDc_files/cDc-351)

*Warning: Foul language on this site*



COMP 6370 – Buffer Overflows – Lecture 8



26

## Dildog's Buffer Overflow Toolkit

- You should be familiar with the following:
  - Intel x86 Assembly, preferably Pentium
  - General Windows System Architecture
  - Know what a URL is.
  - Have a working knowledge of C
- The following tools are suggested to do anything useful:
  - A good hex editor/assembler/disassembler, such as HIEW
  - A realtime debugger, such as SoftICE
  - A few tools that come with Visual C++, DUMPBIN specifically.



COMP 6370 – Buffer Overflows – Lecture 8



27

## Buffer Overflow Attacks can be used by "Script Kiddies"



COMP 6370 – Buffer Overflows – Lecture 8



28