



University of  
Pittsburgh

# Applied Cryptography and Network Security CS 1653



Summer 2023  
Sherif Khattab  
[ksm73@pitt.edu](mailto:ksm73@pitt.edu)

(Slides are adapted from Prof. Adam Lee's CS1653 slides.)

# Announcements

- Please schedule Project Phase 3 Demo with Pratik as soon as possible
- Phase 3 Peer Evaluation Survey is up on CATME
  - Due this Friday
- Homework 9 due this Friday @ 11:59 pm
- Programming Assignment 2 due this Friday
- Project Phase 4 Due on 7/31 @ 11:59 pm
  - Teams must meet with me on or before Thursday 7/27
- Midterm, homework, and phase 1 and 2 grades will be posted tonight

# OS Security - Overview

## OS protection model overview

- Memory protection
- Transitions between user mode and kernel mode

## Integrity of mechanism

## Compromising integrity of mechanism by violating assumptions at

- The OS level: Buffer overflow case study
- The application level: SQL injection case study

# Protection in x86 Processors

x86 processors operate in two distinct modes

- Kernel mode (privileged)
- User mode (unprivileged)

The kernel is typically “small” when compared to the entire OS

- Controls base system functionality
- Provides interface to hardware
- Manages system data structures

Why use a small **trusted** kernel?

- In theory, a small kernel can be rigorously tested and verified
- In practice, it's always good to minimize your trusted computing base
  - Recall Saltzer and Schroeder's **principle of least privilege**

Secure Boot is increasingly common:

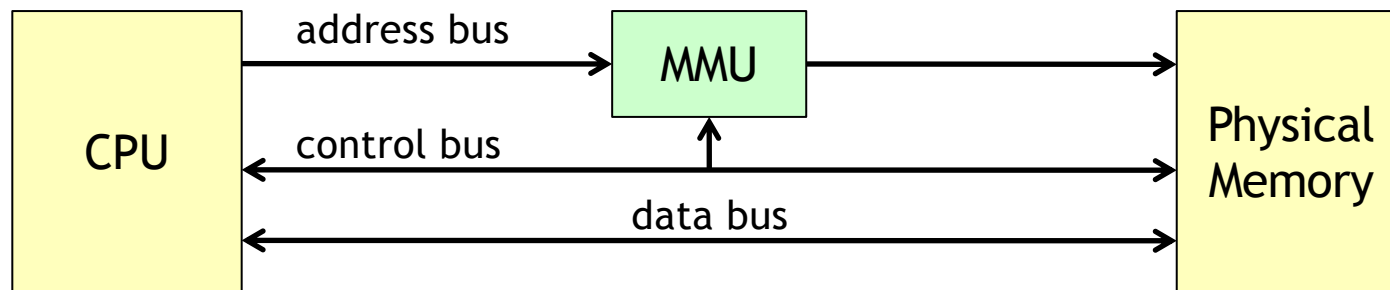
UEFI (United Extensible Firmware Interface), mobile

# One critical job of the OS is memory protection

A user should be **unable** to access memory used by the kernel or other users

- This provides isolation, which is necessary to ensure integrity

How is this enforced?



The memory management unit (MMU) plays the role of the enforcer

- Translates logical memory addresses into physical addresses
- Allows regions of memory to be marked as read only
- Ensures that requests are valid
  - Point to valid physical addresses
  - Are permissible (e.g., not writing to read only memory)

The OS controls the MMU and manages its lookup tables (e.g., page tables)

# System Calls

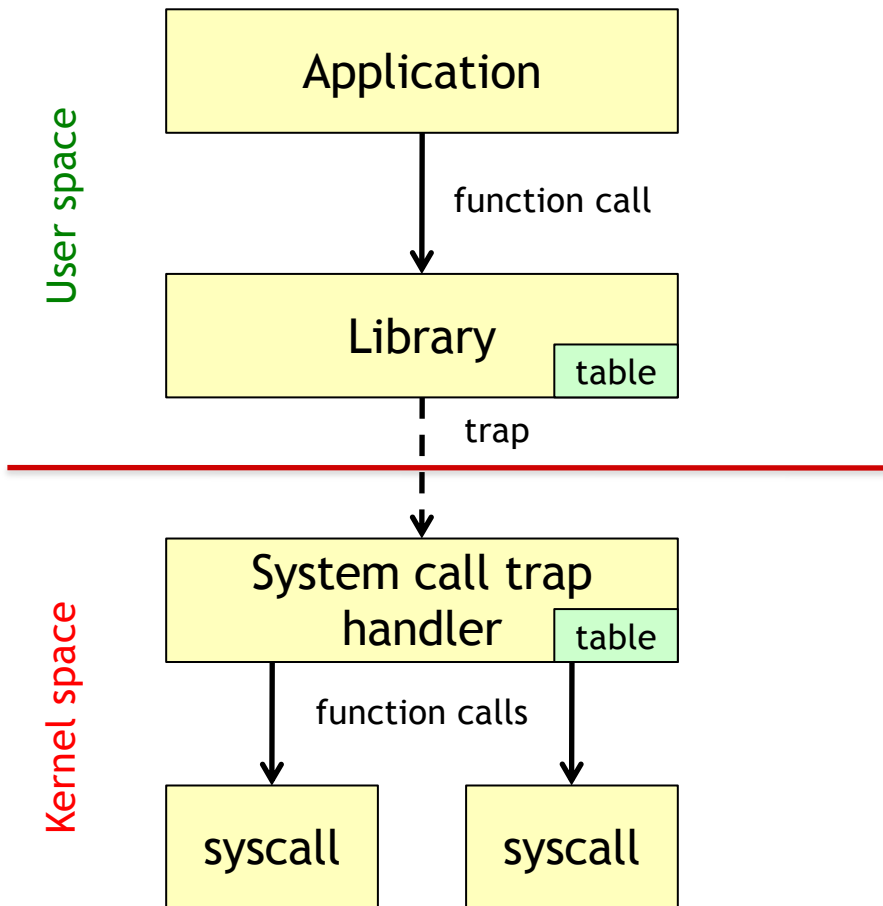
**System calls** allow user space code to access OS services

System calls **transfer control** of the CPU from the user to the kernel

How does this process work?

- User code identifies which kernel service it wants to invoke (e.g., by leaving a code in a specific register)
- Arguments for this service are left on the program stack
- Trap instruction issued
- Trap handler looks up code and gathers any arguments from the stack
- A function call is made

Recall from your OS class that the above process is part of a **context switch**



# OS security depends on integrity of mechanism

The protection mechanisms used by the OS are designed to narrow the interface through which users can access protected data

In order to ensure that the system remains secure, we need to ensure that

- The principle of **complete mediation** is followed
- “Trusted” code is actually trusted

There is also an **implicit assumption** that user processes behave as expected by the users who invoke them

As a result, compromising the security of an OS involves **violating** the **integrity** of its enforcement mechanisms



# How can we violate the OS's assumption of integrity of mechanism?

**Trivial attack:** Extend the Trusted Computing Base (TCB) by installing a malicious device driver

- e.g., Why not write a keyboard driver that's also keystroke logger?
- Drivers are trusted, and have complete access to kernel data structures
- This is a viable attack method, but its uses are limited

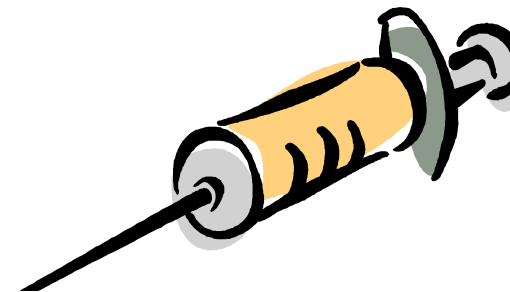
A better idea is to make an **existing** trusted process behave badly

*Approach 1:* Kernel Level



Use a buffer overflow to subvert the kernel's context switch process

*Approach 2:* Application Level



Violate application-level constraints via SQL injection



# Buffer overflow is one of the oldest tricks in the book

In fact, buffer overflow was one of the attack vectors exploited by the Morris worm back in 1988!

**Informally**, a buffer overflow works by providing an extremely long input to a user program that causes system data structures to be overwritten

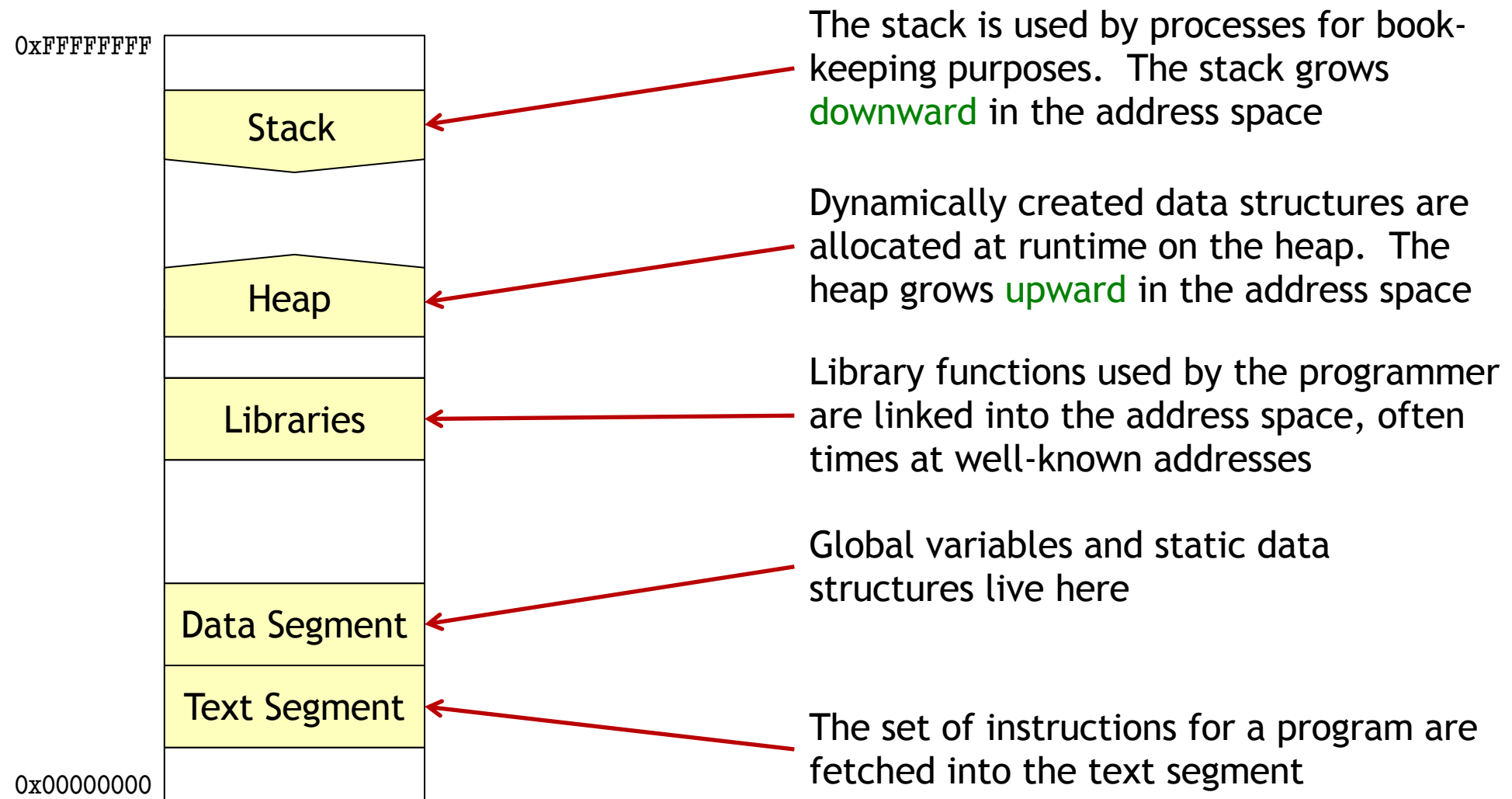
The result of this is that control is transferred to code injected by the attacker, instead of being transferred back to the parent process

This attack vector is best used against root-level processes

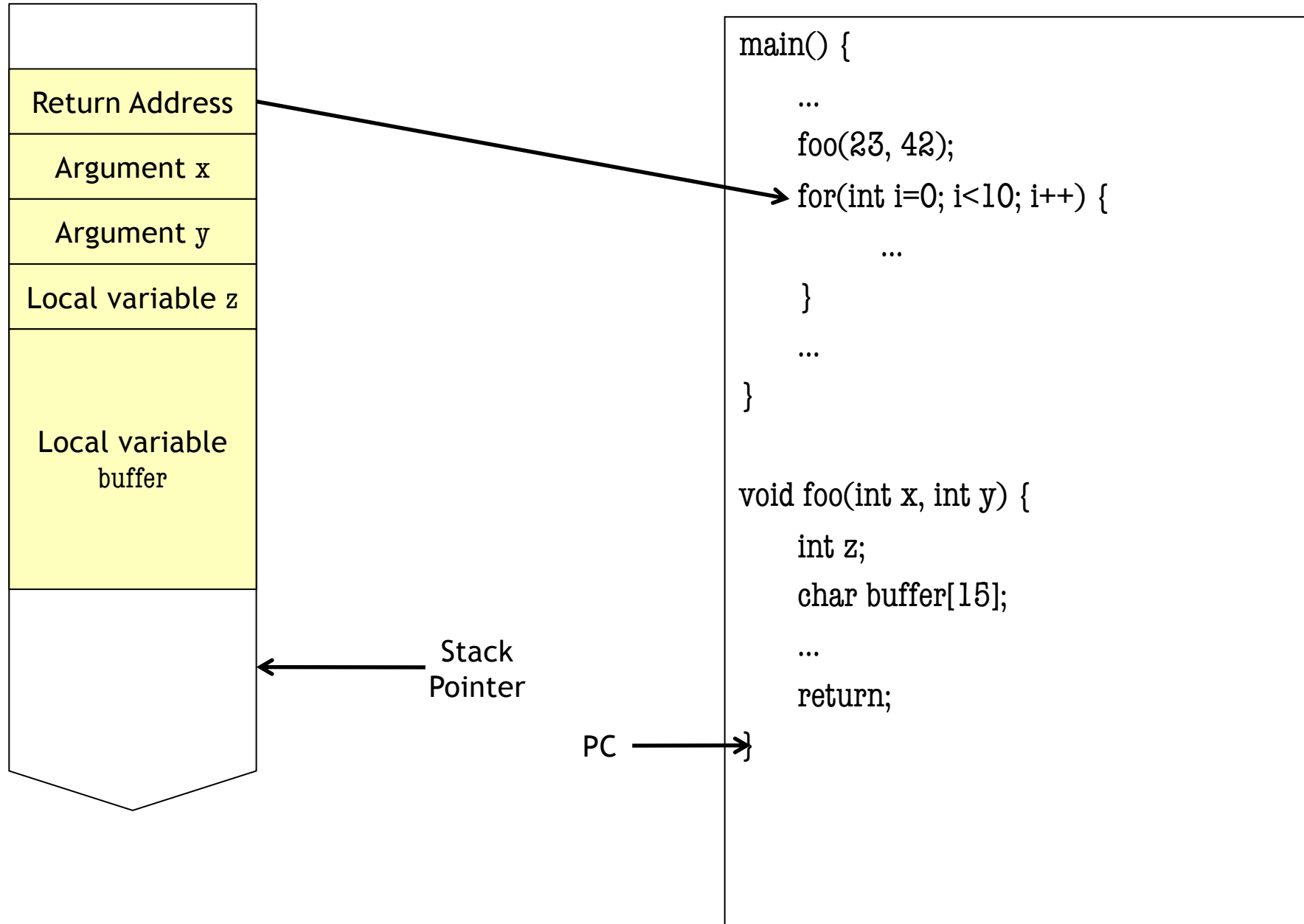
- Why? The attacker's code will run as root!
- This provides a way for the attacker to “trick” the system into granting them privileged access

# Understanding how buffer overflows work means understanding system memory management

Usually, a process's **address space** is a chunk of virtual memory accessed as an array of bytes



# How does the stack work?



# Smashing the stack

Aleph One, Smashing the Stack for Fun and Profit, Phrack Issue 49, Nov. 1996.

In a stack smashing attack, the attacker's goal is to:

- Inject exploit code onto the stack
- Overwrite the return address pointer to transfer control into the exploit code, rather than the calling function

**Result:** If the exploited code was running as an administrator, the attacker's code executes with administrative privileges!

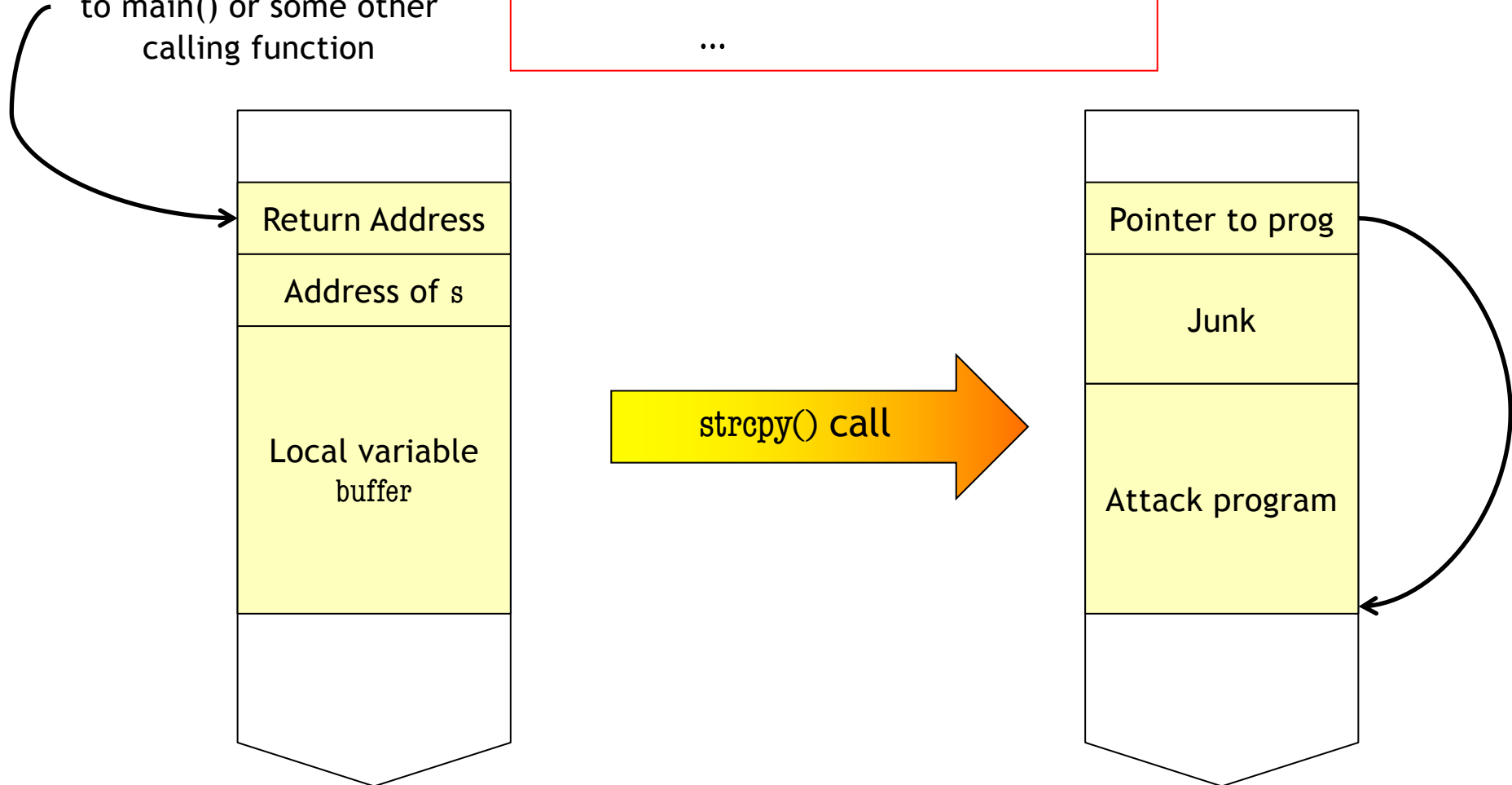
- Unix servers often run as root
- Most Windows users log in with administrative privileges

This can often be accomplished using unsafe string operations in C (e.g., strcpy)

# For example, consider the following program fragment

```
void bar(char *s) {  
    char buffer[100];  
    strcpy(buffer,s);  
    ...  
}
```

Presumably, this returns  
to main() or some other  
calling function



# How can we craft an exploit for the overflow?

Often our exploit code is just a few system calls

**Caveat:** Writing good **shellcode** is a bit of an art

- The attacker needs a good understanding of memory layout
- Code needs to be location independent
- If the code is exploiting C string functions, it cannot include any 0x00 bytes

Assuming that we can do that (see [MetaSploit project](#)) we must still answer two critical questions:

1. Where is the buffer and how big is it?
2. Where is the return address?

In reality, we only need to have **approximate** answers to these questions!

# NOP sleds help us guess the program's start location on the stack

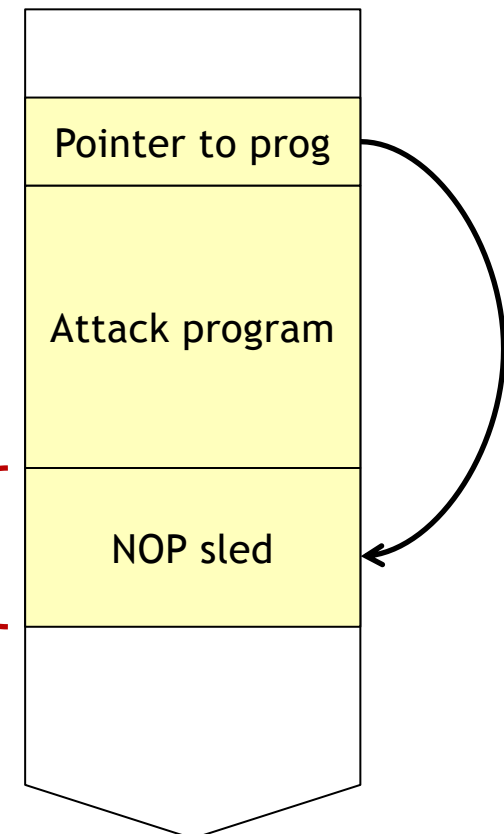
Most CPUs support a *No Operation* instruction of some sort

Rather than starting our attack program at the beginning of our buffer, we can pad it out with a bunch of NOP instructions

This is **good**, because the overwritten return address does not need to point exactly to the location of our exploit code!

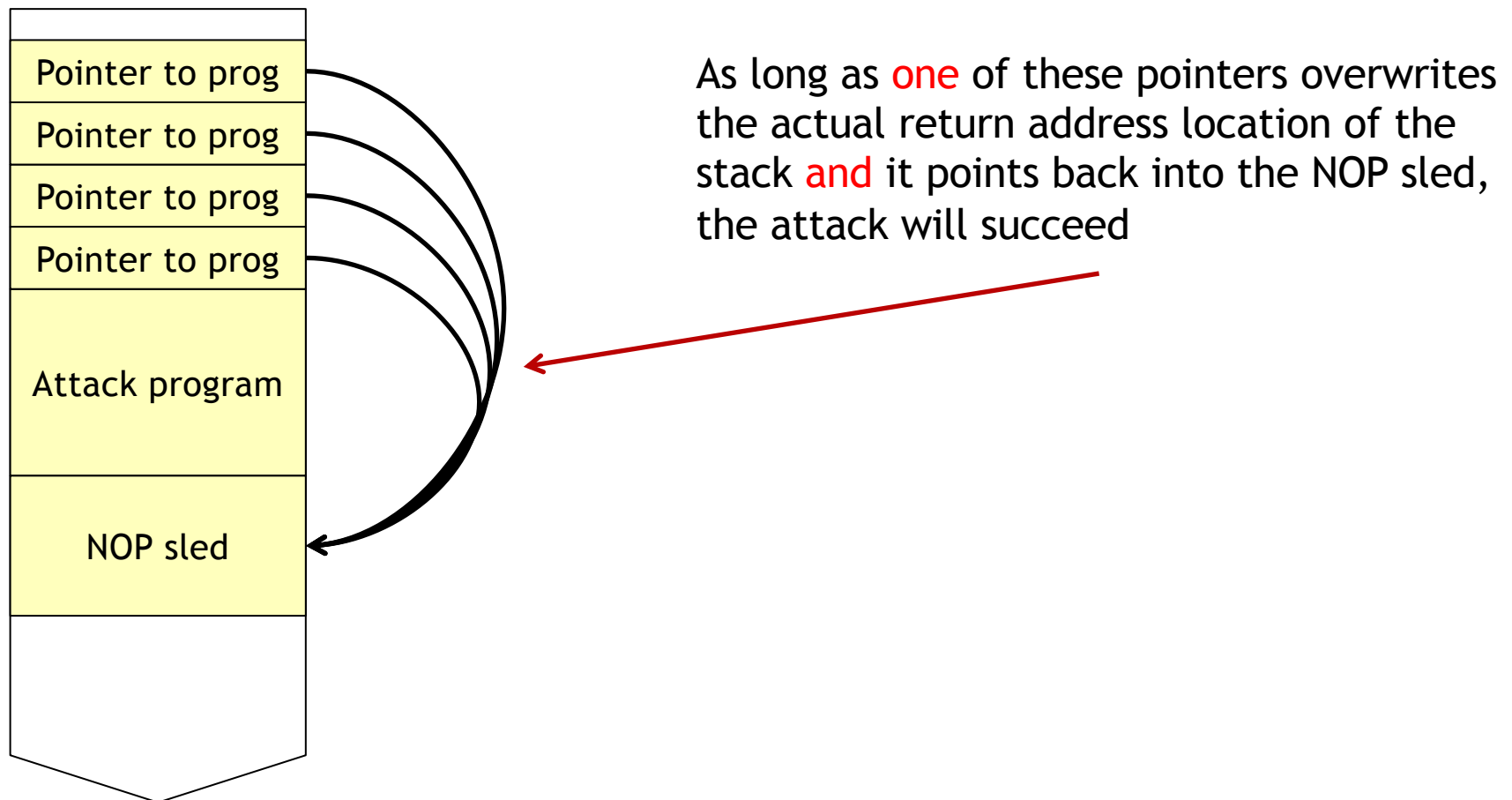
We can point anywhere in here

Unfortunately, we still need to know the **exact location of the return address**, which can be tricky...



# We can also guess the location of the return address on the stack

How? Simply inject **several** new return addresses onto the stack!





# So... How do we defend against buffer overflows?

One way to stop buffer overflows is to use **non-executable stacks**

- If nothing can be executed from the stack, the exploit will fail
- Unfortunately, this is hard to implement in practice, as legacy compilers tend to like putting executable code in the stack
- A variation of this approach involves making writable memory pages non-executable

Another defense mechanism is the use of **canary words**

- A canary word is a random word of data written just before the return address on the stack
- Odds are, if the return address is overwritten, so is the canary word
- See Crispin Cowan's *StackGuard* for more information



check Laundry Overflow animation at

<https://oercommons.org/courses/laundry-overflow-an-interactive-animation-analogy-for-buffer-overflow>

# So... How do we defend against buffer overflows?

Some researchers have proposed the use of **address space randomization** (ASR) to protect against stack smashing [1,2]

- **Intuition:** Shuffle the locations of variables and static data stored on the stack randomly each time the program is executed
- If the buffer is overflowed, there's really no way\* to tell what data structures will be overwritten
- Unfortunately, researchers have shown that certain variations of this approach can be defeated [3]

Another way to stop stack smashing is to use a type-safe programming language (e.g., Java or C#) ☺

[1] S. Bhatkar, R. Sekar, and D. DuVarney. Efficient Techniques for Comprehensive Protection from Memory Error Exploits,” 14<sup>th</sup> USENIX Security Symposium, August 2005.

[2] <http://pax.grsecurity.net>

[3] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Bohen, “On the Effectiveness of Address-Space Randomization,” 11<sup>th</sup> ACM Conference on Computer and Communications Security, pages 298-307, Oct. 2004.

# Stack smashing has a number of variants

If an attacker cannot launch a “traditional” stack smashing attack, there are other avenues that she could follow

**Heap smashing** is like stack smashing, but corrupts heap data

- There is much more variety across systems with how the heap is managed
- Attacks are difficult to construct, and typically are specific to a specific machine and/or configuration

**Return to libc** attacks jump to existing library code, not attacker code

- The attacker first correctly places the arguments on the stack
- Then sets the return pointer to jump to pre-loaded library function (e.g., the libc system() function)

**Return oriented programming** generalizes return to libc

- Generate a Turing complete library of computational “widgets” from the libc library (or any other library)
- Inject an attack vector that weaves widgets in some desirable way
- The effect is running arbitrary code **without** injecting executable code!

# Buffer Overflow Wrap-Up

In a stack smashing attack, the attacker alters kernel data structures to make a process behave in an unexpected manner

If the process that is attacked is running with elevated privileges (e.g., a server running as root), then the attacker can do all sorts of fun things

- Set up tunnels into the system
- Alter the password file
- Install rootkits to gain long-term control of the system
- Delete files
- ...

Clearly, this constitutes a violation of integrity of mechanism

# What about violating constraints within an application?

Sometimes, we don't need to subvert the entire OS to wreak havoc.

For example, many times applications have their own security mechanisms in place

- These constraints control the functionality exposed to different users
- Enforced by the application, not the OS
- In this case, compromising the application is all that is needed

## *Case Study:* SQL Injection



# SQL injection attacks are specific to application servers with database back ends

These applications serve **dynamic content** based upon queries that are parameterized by user input

*Example:* An online phone book application

Steph 555-1234  
Chris 555-4245  
Sherif 555-3211  
... ..

Name: Sherif  
Password: l3tMe1n

Submit

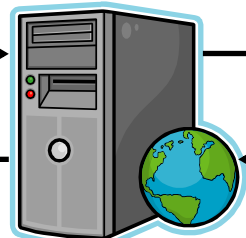
```
SELECT * FROM phonebook WHERE  
username = 'Sherif' and password =  
'l3tMe1n'
```



Web Browser

Form Data

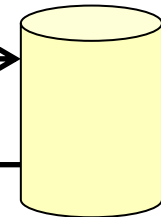
Web Page



Application Server

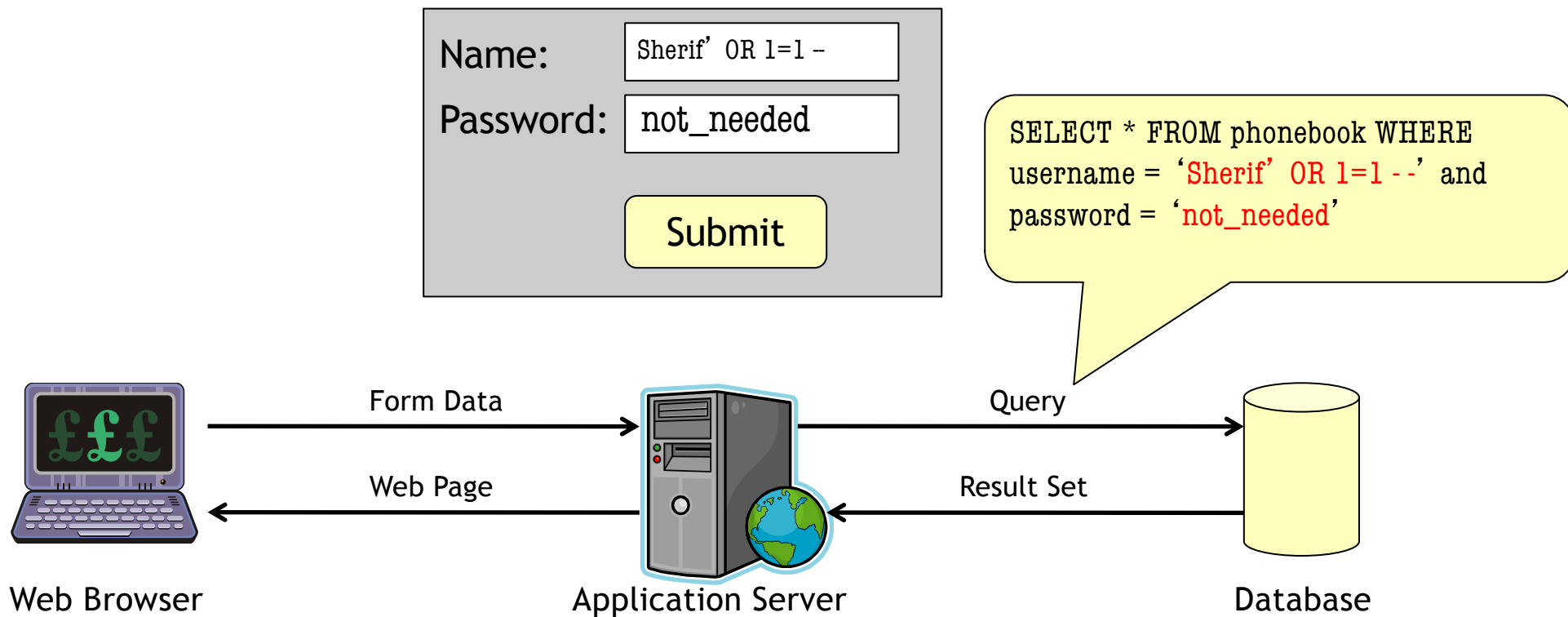
Query

Result Set



Database

# What could possibly go wrong with such a simple application?



Note that:

- The test `1=1` is always true!
- Also, `--` denotes a comment in SQL, so the password check is bypassed!

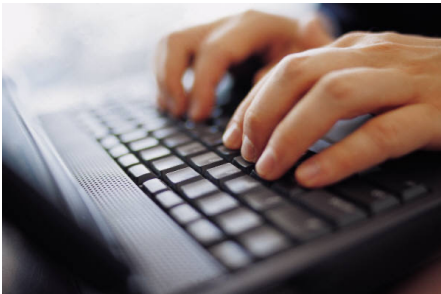
**Result:** All rows are displayed!

# SQL injection can be used for all kinds of attacks...



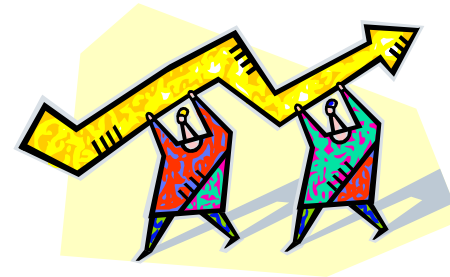
Gaining access to the system

- Perform an INSERT on the “Users” table



Modify data in the database

- Drop tables
- Change salaries
- Etc.



Perform privilege escalation attacks

- Stealing passwords
- Inserting rights
- Etc.



# Clearly, this is much easier than crafting a buffer overflow!

The real difficulty lies in ascertaining the structure of the target database, but even this is not so difficult... So how we defend ourselves?

One solution is to sanitize database inputs

- For example, ' → \', " → \", and \ → \\
- Unfortunately, this doesn't always work...
  - `SELECT * FROM salary WHERE id = 23 OR 1=1`

Check input field validity

- Fields typically have fixed vocabularies
- Make sure illegal characters don't appear
- Unfortunately, this isn't always possible

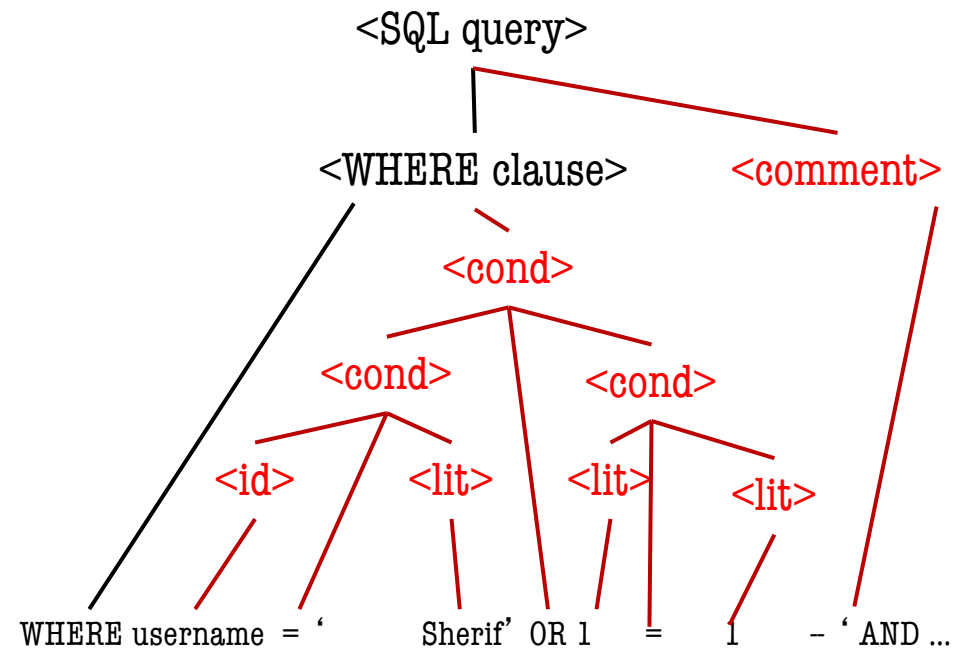
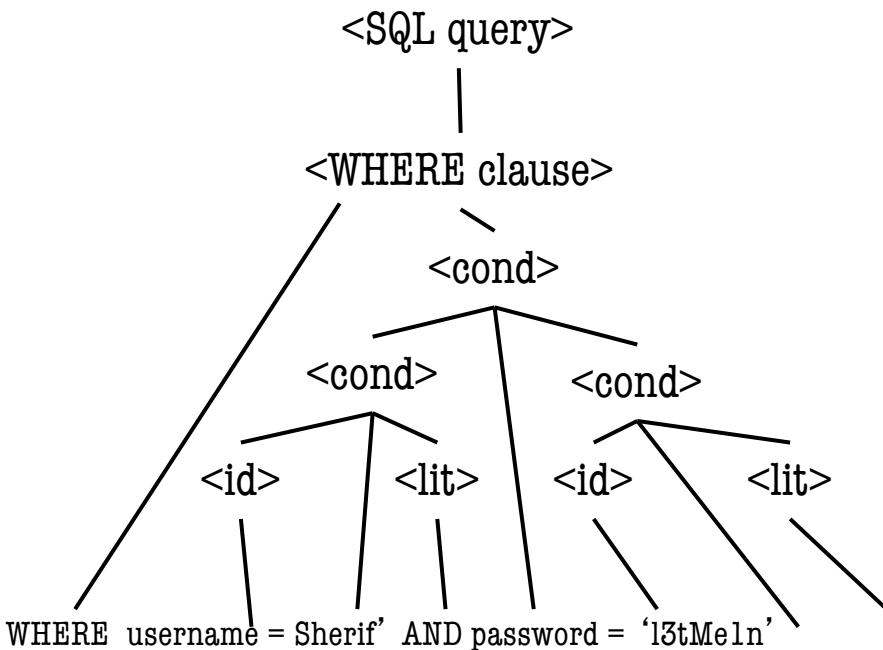
Place length limits on the allowed inputs to HTML Form fields

*These are all basically hacks...*

# A solution proposed by Bandhakavi et al. uses parse tree comparison to detect SQL injection

**Intuition:** The parse tree for an SQL injection query is almost always different than the parse tree for the programmer intended query

**Example:**



Sruthi Bandhakavi, Prithvi Bisht, Madhusudan Parthasarathy, V.N. Venkatakrishnan, "CANDID: Preventing SQL Injection Attacks using Dynamic Candidate Evaluations," 14th ACM Conference on Computer and Communications Security (CCS), November 2007.

# The problem that must be addressed is how to infer the programmer intended parse tree for a given query

Bandhakavi et al. solve this problem by generating benign **candidate inputs** for each dynamically generated query

The problem of **how** to generate inputs that are benign and do not alter the program's control flow is undecidable in general...

However, simple heuristics make this possible!

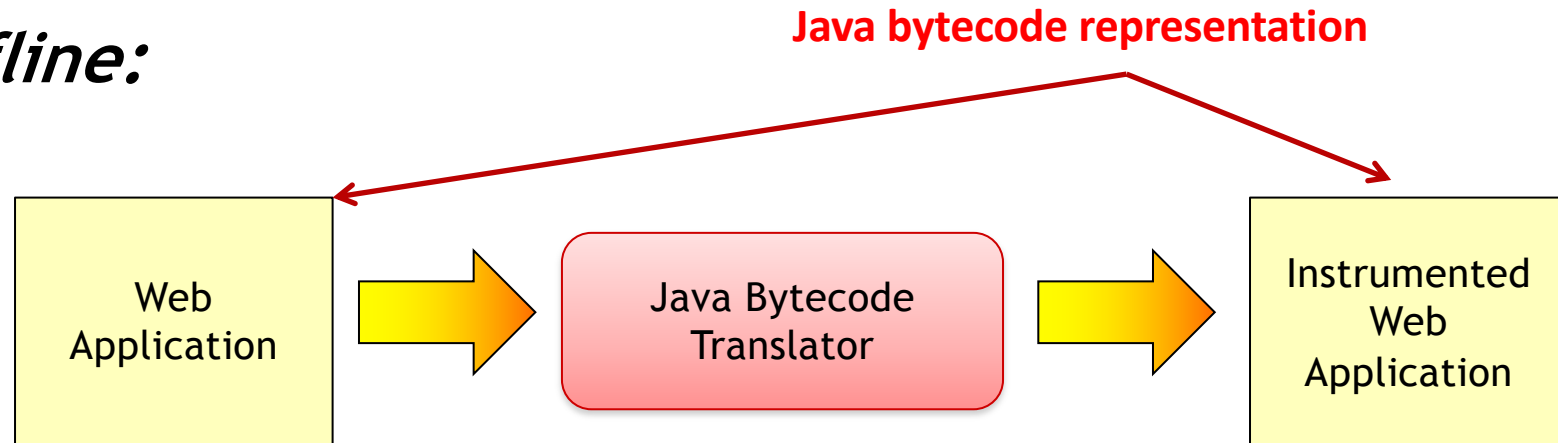
- Replace all integer inputs with "1"
- Replace all string inputs with an equal length string containing only "a"s
- Replace all booleans with "true"
- Make control flow decisions based upon **actual** input only

Parse trees are generated for the actual query and the candidate query

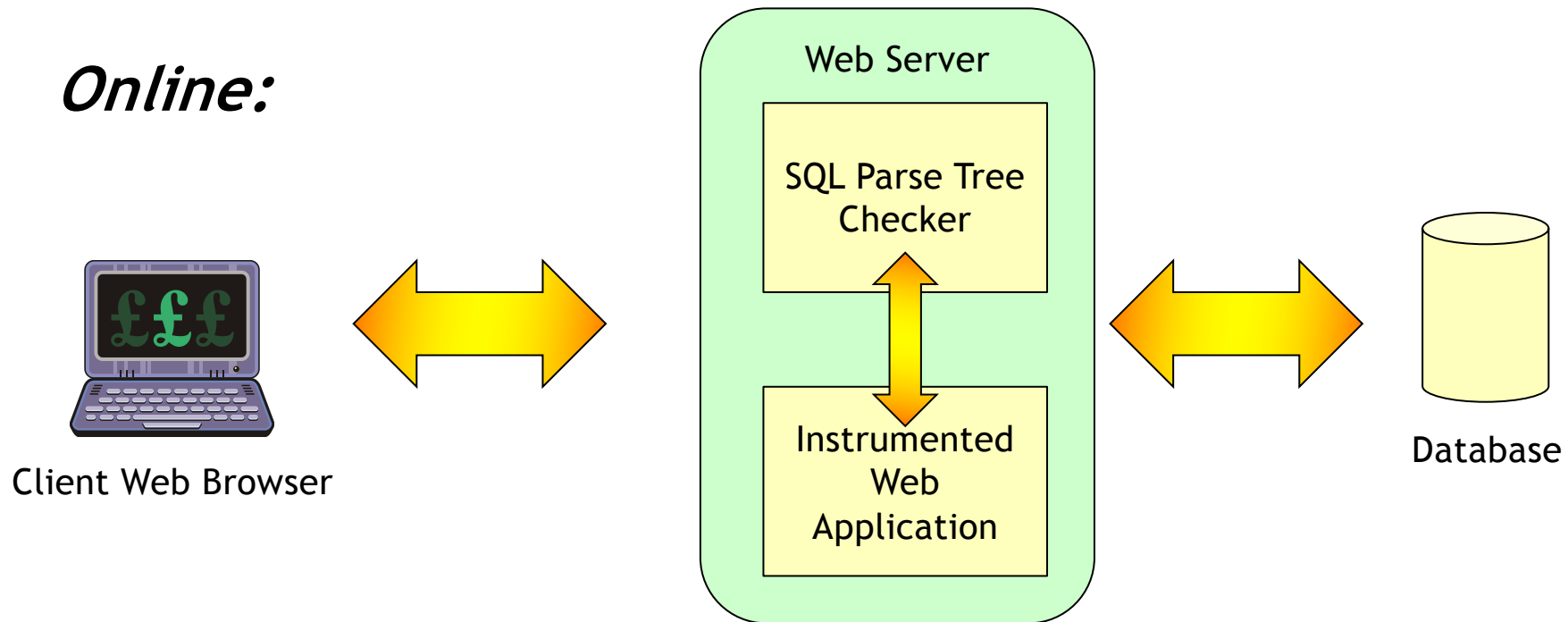
- Actual query is issued only if both parse trees match

# How is the CANDID system set up?

*Offline:*



*Online:*

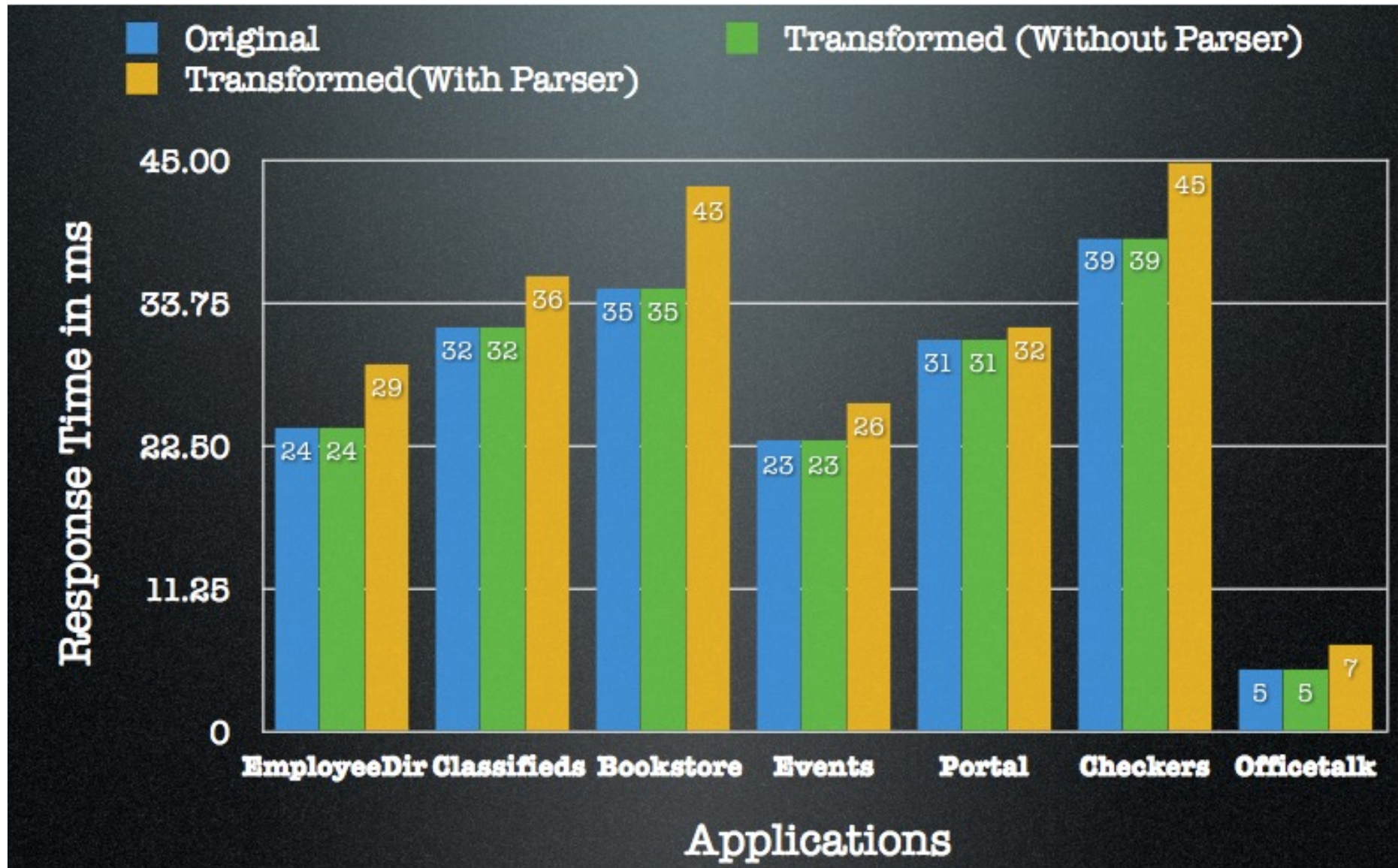


# Bandhakavi's solution performs well on standard database benchmarks

Application	Lines Of Code	Successful Attacks / Attacks Detected	Non-attacks / False Positives	Parse Errors Expected/ Detected
Bookstore	16959	1438 / 1438	2930 / 0	2124/2124
EmployeeDir	5658	1529 / 1529	2442 / 0	3067/3067
Events	7242	1414 / 1414	3320 / 0	2375/2375
Classifieds	10949	1475 / 1475	2076 / 0	3128/3128
Portal	16453	2995 / 2995	3415 / 0	1073/1073
Checkers	5421	262 / 262	7435 / 0	557/557
Officetalk	4543	390 / 390	2149 / 0	4060/4060



# Bandhakavi's solution performs well on standard database benchmarks



# Summary

OS Security relies on separating the **trusted** from the **untrusted**

The assumption of **integrity of mechanism** is central to security

Integrity of mechanism can be violated at the OS or application level

- **OS example:** Buffer overflow
- **Application example:** SQL injection

**Next:** Network security



# So what? A high-value target will always fall eventually

Sometimes compromising one machine/service is just the beginning

- Compromising a service can violate the assumptions of other entities in the system
- Easily bootstrap the compromise of any machine that trusts the compromised service
- Spread to more vulnerable machines

Once integrity of mechanism is violated, all bets are off

Viruses and worms are malware that **propagate**



# Viruses and Worms

Definitions

*Case studies:*

- The Brain virus
- The Morris worm
- Code Red

Into the future

# Malicious logic is a set of instructions that cause an organization's security policy to be violated

One common type of malicious logic is the **Trojan horse**, which is a program that has a well-known **overt** effect, and an unknown **covert** effect.



## **Example:** NetBus

- Allows an attacker to remotely administer a Windows NT box
- Remote admin code was bundled with games/“fun” programs

Not all Trojan horses are so easy to spot...

## **Example:** Thompson's login program

- Modification to UNIX login program that accepted a default password
- Modify compiler to insert default case when compiling the login program
  - Backdoor **is not** visible in login source code
  - Backdoor code **is** visible in compiler code
- Modify compiler to insert above code if the compiler is being compiled
- Install malicious compiler binary, and benign compiler source
  - Backdoor invisible in **all** source code

# The problem with a Trojan horse is that you need to convince the victim to install the host program

A **computer virus** is a program that **inserts itself** into one or more files and then performs some action



The term computer virus was coined by Fred Cohen and his advisor Len Adelman at USC in 1983

- Cohen's thesis is one of the few theoretical results regarding viruses
- **Key result:** No algorithm can detect computer viruses precisely
- This is why virus scanners are largely heuristic...

Why are viruses called viruses?

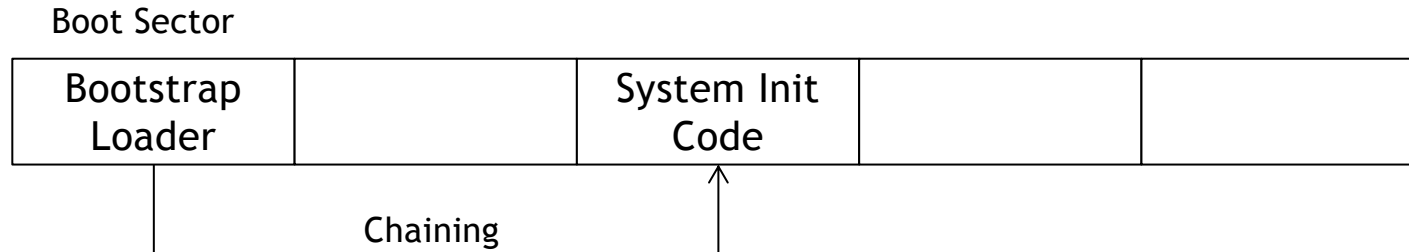
Because they self-replicate by attaching themselves to host programs!

In the days before widespread Internet availability, virus writers had to get creative to enable the spread of a virus beyond one system

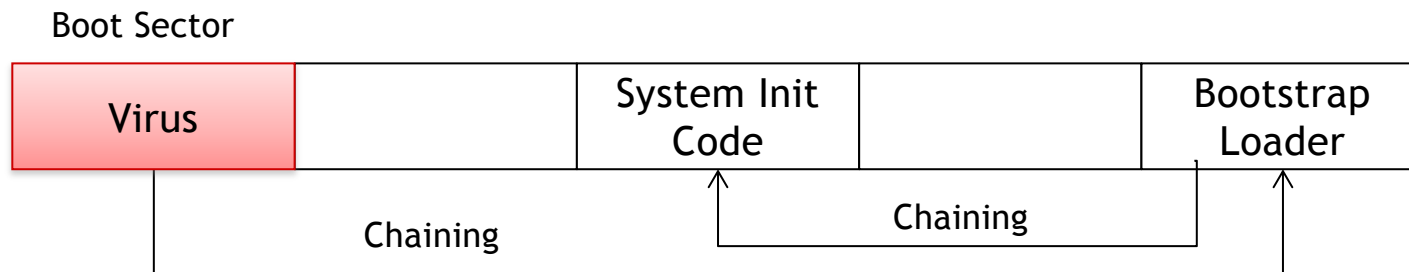
# Some viruses spread by infecting a drive's boot sector

How does the boot process work?

- System firmware reads a specific disk sector (the **boot sector**) into memory
- The system then jumps to that memory location
- The boot program then begins loading the OS



A **boot sector virus** hijacks this process to facilitate its spread



# Case Study: The Brain virus

```
PC Tools Deluxe 84.22
Disk View/Edit Service
Path=A:
Absolute sector 00000000, System BOOT

Displacement  Hex codes  ASCII value
0000(0000)  FA E9 4A 01 34 12 00 07 14 00 01 00 00 00 00 20  -0J04: 07 0
0016(0010)  20 20 20 20 20 20 57 65 6C 63 6F 6D 65 20 74 6F  Welcome to
0032(0020)  20 74 68 65 20 44 75 6E 67 65 6F 6E 20 20 20 20  the Dungeon
0048(0030)  20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0064(0040)  20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0080(0050)  20 20 63 23 20 31 39 38 36 20 42 61 73 69 74 20
0096(0060)  26 20 41 6D 6A 61 64 20 28 70 76 74 29 20 4C 74  (c) 1986 Basit
0112(0070)  64 2E 20 20 20 20 20 20 20 20 20 20 20 20 20 20  & Amjad (pvt) Lt
0128(0080)  20 42 52 41 49 4E 20 43 4F 4D 50 55 54 45 52 20  d.
0144(0090)  53 45 52 56 49 43 45 53 2E 2E 37 33 30 20 4E 49  BRAIN COMPUTER
0160(00A0)  5A 41 4D 20 42 4C 4F 43 4B 20 41 4C 4C 41 4D 41  SERVICES..730 NI
0176(00B0)  20 49 51 42 41 4C 20 54 4F 57 4E 20 20 20 20 20 20  ZAM BLOCK ALLAMA
0192(00C0)  20 20 20 20 20 20 20 20 20 20 20 4C 41 48 4F 52  IQBAL TOWN
0208(00D0)  45 2D 50 41 4B 49 53 54 41 4E 2E 2E 50 48 4F 4E  LAHORE
0224(00E0)  45 20 3A 34 33 30 37 39 31 2C 34 34 33 32 34 38  E-PAKISTAN..PHON
0240(00F0)  2C 32 38 30 35 33 30 2E 20 20 20 20 20 20 20 20  E :430791,443248
,280530.

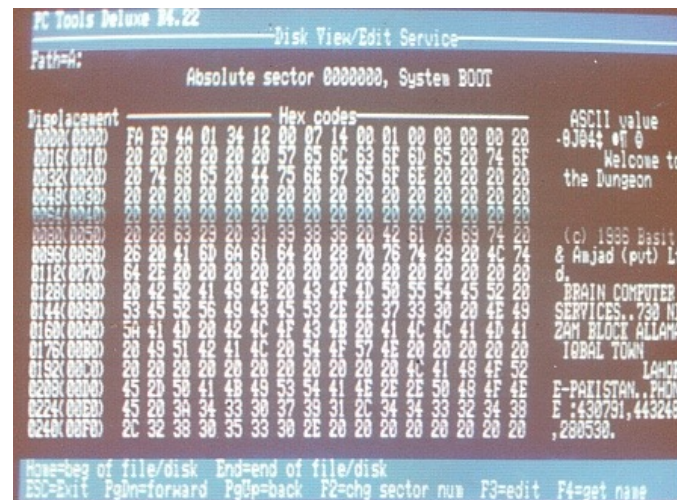
Howe=begin of file/disk  End=end of file/disk
ESC=Exit  PgDn=forward  PgUp=back  F2=chg sector num  F3=edit  F4=get name
```

# The Brain virus was one of the first, and most carefully studied, computer viruses

Brain was a boot sector virus that originated in Pakistan in 1986

Two brothers, Basit and Amjad Farooq Alvi, supposedly wrote Brain to protect medical software that they wrote from copyright infringement

However, the Brain virus contained no malicious payload and actually advertised the brothers' contact information!



```
PC Tools Deluxe 04.22      Disk View/Edit Service
Path=\\:
      Absolute sector 00000000, System BOOT

Displacement  Hex codes  ASCII value
0000(0000)  FA E9 4A 01 34 12 00 07 14 00 01 00 00 00 00 20  -0J94: of 0
0010(0010)  20 20 20 20 20 20 57 65 6C 63 6F 6D 65 20 74 6F  Welcome to
0020(0020)  20 74 6F 65 20 44 75 6E 67 65 6F 6E 20 20 20 20  the Dungeon
0030(0030)  20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0040(0040)  20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0050(0050)  20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0060(0060)  20 20 63 23 20 31 33 38 36 20 42 61 73 65 74 20  (c) 1986 Basit
0070(0070)  26 20 41 6D 6A 61 64 20 20 70 76 74 20 20 4C 74  & Amjad (pvt) Lt
0080(0080)  64 2E 20 20 20 20 20 20 20 20 20 20 20 20 20 20  d.
0090(0090)  20 42 52 41 49 4E 20 43 4F 41 50 55 54 45 52 20  BRAIN COMPUTER
00A0(00A0)  53 45 52 56 49 43 45 53 2E 2E 37 33 30 20 4E 49  SERVICES..730 NI
00B0(00B0)  54 41 4D 20 42 4C 4F 43 4B 20 41 4C 4C 41 4D 41  2AM BLOC2 ALLAMA
00C0(00C0)  20 49 51 42 41 4C 20 54 4F 57 4E 20 20 20 20 20  IQBAL TOWN
00D0(00D0)  20 20 20 20 20 20 20 20 20 20 20 20 41 41 49 4F 52  LAHORE
00E0(00E0)  45 20 50 41 49 49 53 54 41 4E 2E 2E 50 48 4F 4E  E-PAKISTAN..PHON
00F0(00F0)  45 20 34 34 33 30 37 39 31 2C 34 34 33 32 34 38  E :430791.443248
0100(0100)  2C 32 38 30 35 33 30 2E 20 20 20 20 20 20 20 20  ,280530.

Howe=begin of file/disk  End=end of file/disk
ESC=Exit  PgDn=forward  PgUp=back  F2=chg sector num  F3=edit  F4=get name
```

It is suspected that Brain was really a cute gimmick to draw attention to their business!



# How did the Brain virus work?

When a system boots from an infected disk:

1. The virus **loads** itself in upper memory
2. It then **resets** the user accessible memory boundary to just below itself
3. Brain then mangles the system interrupt table
  - Interrupt 19 (disk read) is reset to point to the Brain code in memory
  - Interrupt 6 (unused) is then set to point to the old code for interrupt 19

Whenever a disk read occurs

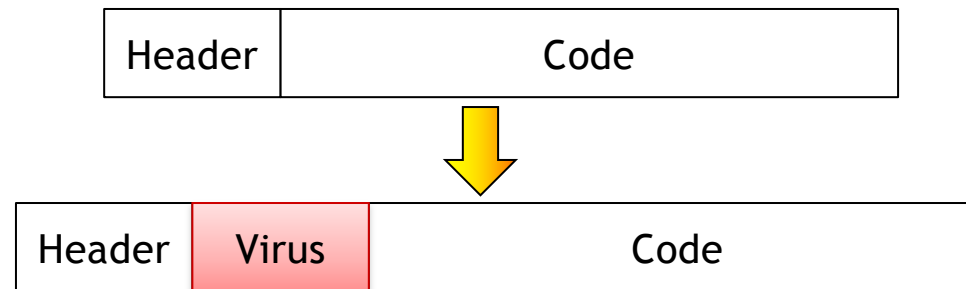
1. Interrupt 19 is triggered and transfers control to the virus
2. If the disk being read is **not** yet infected, the virus infects it
3. The virus then triggers interrupt 6 to allow the disk read to occur

Although Brain contained no malicious payload, it was used as a template for several more destructive viruses

---

**Question:** How do you think that users detected this virus?

# There are many other types of viruses...



## *Executable viruses*

- Attach to executable code
- Invoked when code is invoked
- *Example:* Jerusalem



## *TSR viruses*

- TSR = Terminate and stay resident
- Virus stays in memory even after host process terminates (syscall interception)
- *Examples:* Brain and Jerusalem



## *Macro viruses*

- Infect documents, not executables
- Not bound by system architecture
- *Examples:* Melissa

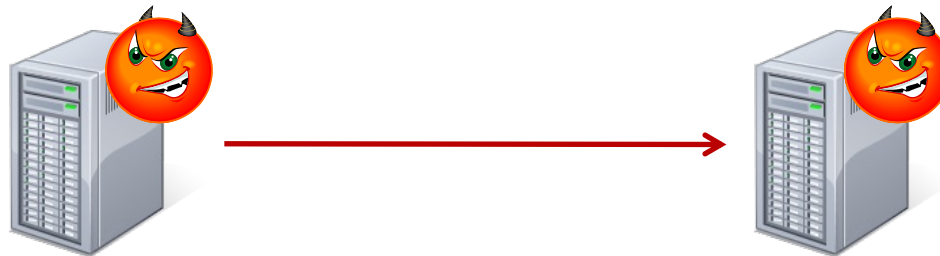


# How fast can viruses spread?

In the early days, the speed with which a virus could spread was limited to the speed of human interactions

- Trading floppy disks
- Sharing spreadsheets or other documents at work
- Installing Trojan programs passed along by a friend
- ...

The growing prevalence of the Internet during the late 80s and early 90s gave rise to **computer worms**, which are viruses capable of spreading themselves across many machines



**The result:** Much speedier infection rates!

# Case Study: The Morris worm

The first major Internet worm was released in 1988

- Written by Robert Tappan Morris
- Launched around 5:00 PM on 2<sup>nd</sup> November 1988
- Originated at Cornell University

The worm was purportedly written to assess the size of the Internet and had no malicious payload

Unfortunately, unbounded replication of the worm brought many systems down

This worm used many techniques that we have already studied...



Eugene H. Spafford, "The Internet Worm: Crisis and Aftermath," Communications of the ACM, 32(6): 678-687, June 1989.

# How did it spread?

Rather than reinvent the wheel, the Morris worm leveraged three **well-known vulnerabilities** to enable its spread across machines running Berkeley and Sun UNIX

## *Method 1:* fingerd

- fingerd provides a lookup service for users' public contact information
- The version of fingerd running on many systems had a buffer overflow due to the use of the unchecked `gets()` routine

## *Method 2:* A bug in sendmail

- sendmail is a popular e-mail routing program
- If operating in DEBUG mode, sendmail allows system commands to be transmitted over SMTP

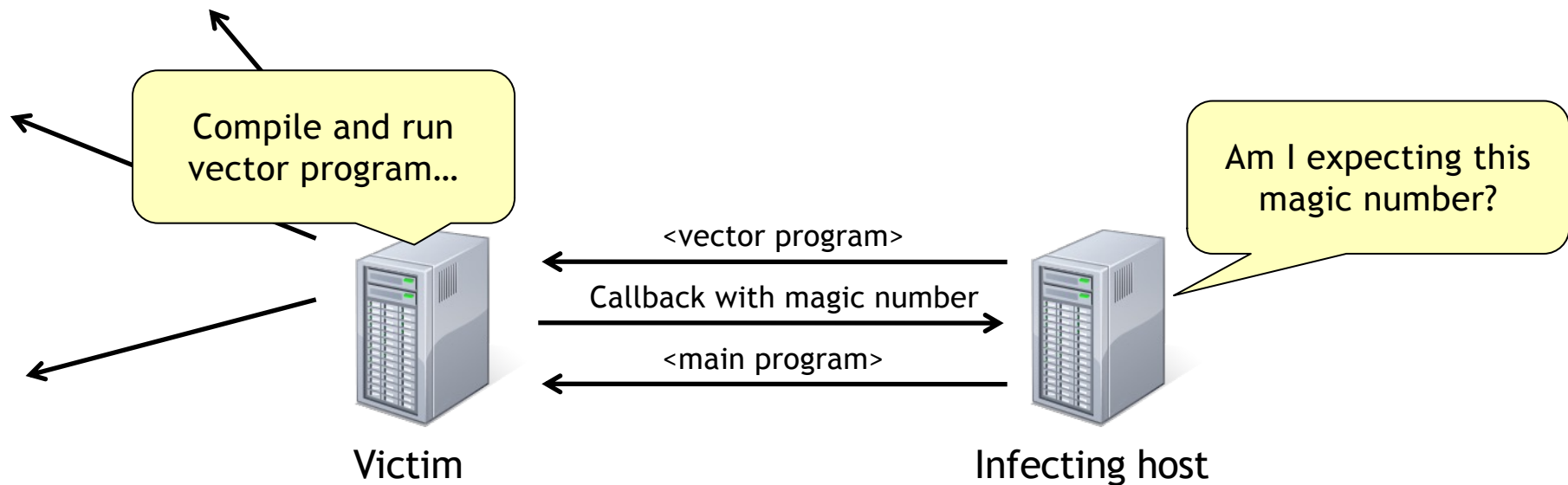
## *Method 3:* Weak password security

- Many passwords are weak and can be broken with simple guessing
- The rsh protocol allows trusted users/hosts to **bypass** authentication

# How did an infection proceed?

The worm consisted of **two programs**: a short vector (i.e., infection) program, and the main program

The vector program was 99 lines of C code, transferred using one of the previous three known exploit techniques

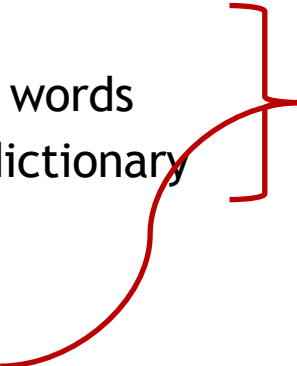


If the (binary) attack programs would not run on the victim, the vector would delete everything to cover its tracks

# What did the main program do?

The main program first gathered information about network interfaces and hosts on the local network, which was randomized to provide a set of hosts to attack

The main program then entered a simple state machine

- Read `/etc/hosts.equiv` and `/.rhosts` to look for trusted hosts to add to table
  - Try to break user passwords using simple choices
  - Try to break user passwords using a dictionary of 432 words
  - Brute force user passwords using entire UNIX online dictionary
- 

**If a password was broken, it was used in conjunction  
with rsh to infect other hosts**

Each state was run for short periods of time, between which the main program attempted to infect other hosts in the list to attack

# The Morris worm actually took steps to prevent overloading a particular host

The worm would periodically check for copies of the worm running on the same host by connecting to a predetermined TCP socket

If another worm was found, one of the two would randomly quit

However, this didn't work terribly well...

- Race conditions in the code sometimes prevented worms from connecting
- Furthermore, 1 in 7 worms became immortal to prevent fake kills

**Result:** Many hosts had several copies of the worm running

---

It is interesting to note that the worm occasionally forked itself

- This gave the worm a new PID to prevent detection
- Also prevented the worms priority from downgrading over time

# Outcomes of the Morris worm

Experts think that upwards of 6,000 hosts were infected with the Morris worm, though this number is an estimate

The Morris worm brought network security to the forefront

- More regular software patching/updating
- Broader proliferation of shadow password files
- Inspired much intrusion detection research

In response to this incident, DARPA funded CERT/CC to monitor computer vulnerabilities, and manage incident response

Robert T. Morris

- **Was** the first person convicted under the 1986 Computer Fraud and Abuse Act
- **Is now** a professor at MIT 😊

# Case Study: Code Red

Although the Morris worm was the first widespread worm, it was certainly not the last!

Software is being developed at an astounding rate

- Software is getting more complex
- Higher complexity leads to more bugs
- More bugs means more potential for exploits

The Code Red worm (v2) was launched on July 19, 2001

- Exploited a bug in Microsoft IIS server
- Infected over 359,000 hosts in under 14 hours!

Although this worm was released 13 years after the Morris worm, it is structurally very similar...



# How did Code Red work?

Like the Morris worm, Code Red utilized a buffer overflow to propagate

- [illegible]

After a host was infected, it spawned 300-600 threads that would:

- Randomly choose an IP address and attempt to connect
- If success, attempt the above buffer overflow

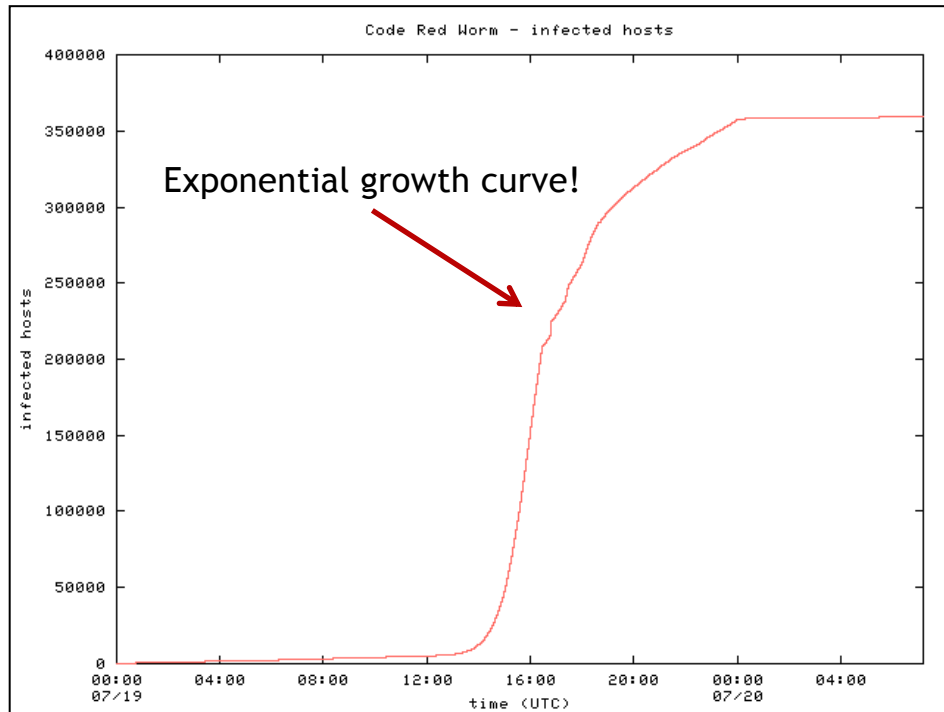
## Code Red's behavior was dependent on the day of the month

- **Day 1 - 19:** Randomly infect other hosts
- **Day 20 - 27:** Carry out a packet-flooding denial of service attack on the IP address of `www1.whitehouse.gov`
- **Day 28 - <end of month>:** Sleep

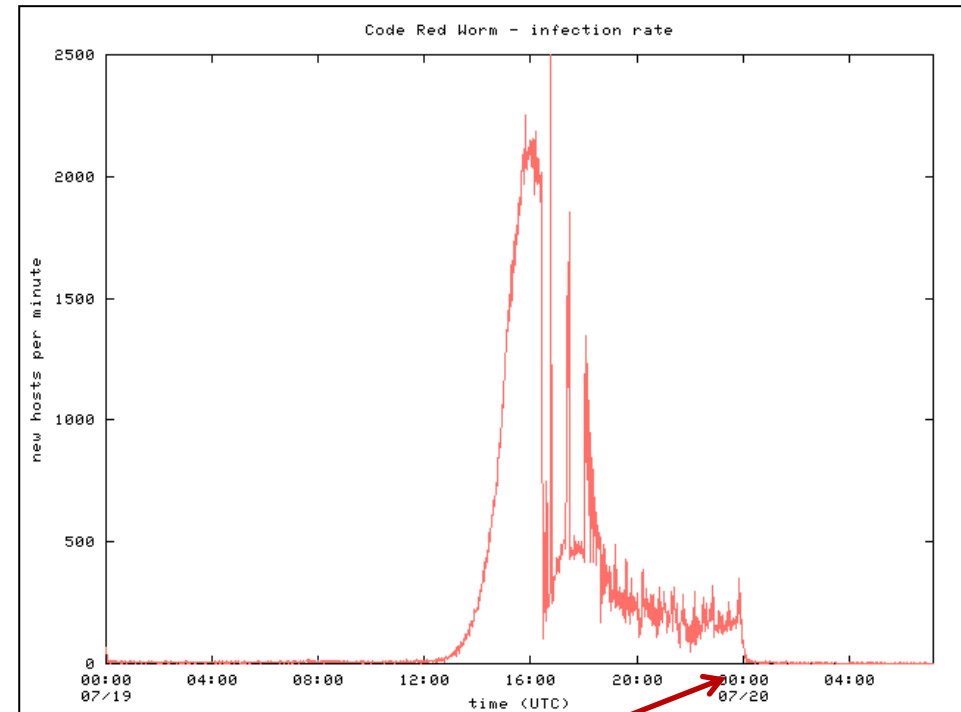
CERT® Advisory CA-2001-19 "Code Red" Worm Exploiting Buffer Overflow In IIS Indexing Service DLL  
<http://www.cert.org/advisories/CA-2001-19.html>

# Code Red spread at an alarming rate

[http://www.caida.org/research/security/code-red/coderedv2\\_analysis.xml](http://www.caida.org/research/security/code-red/coderedv2_analysis.xml)



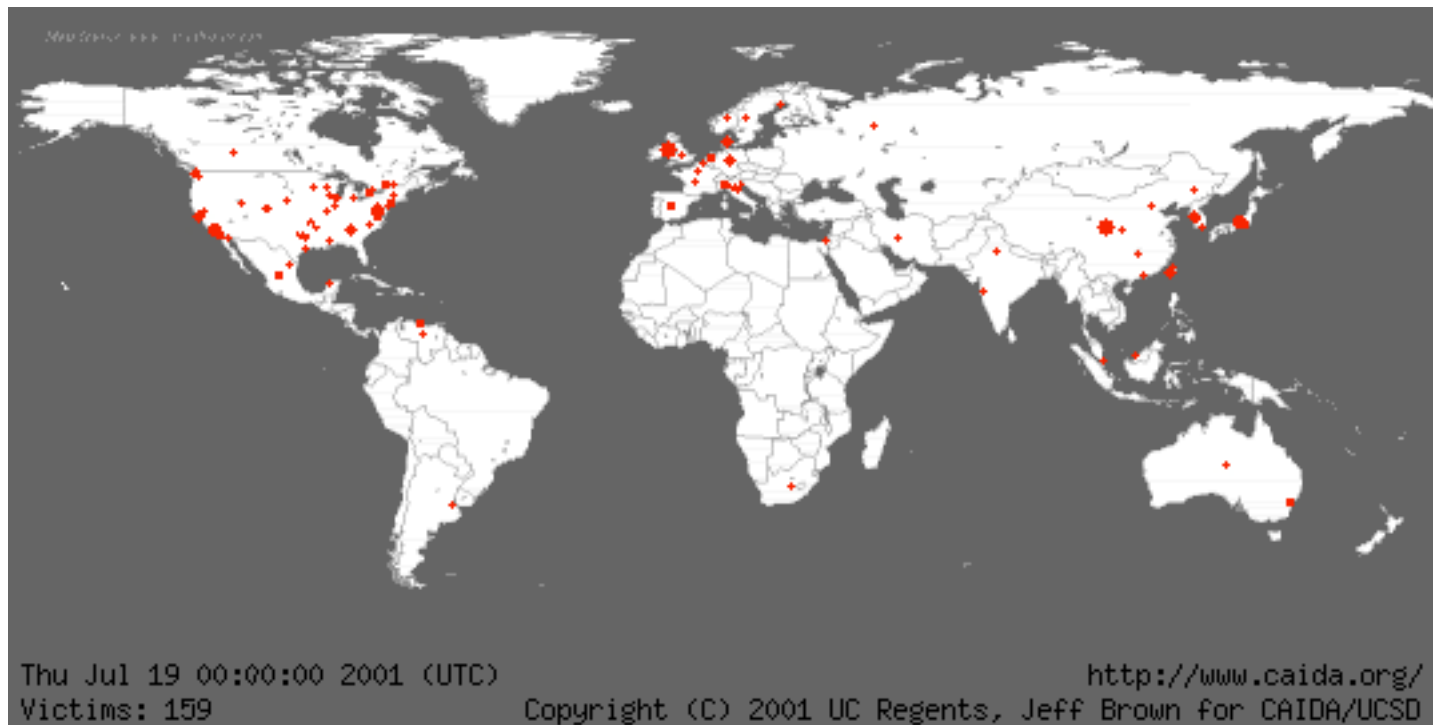
Only 13 hours were needed to infect over 359,000 hosts!



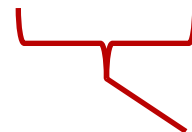
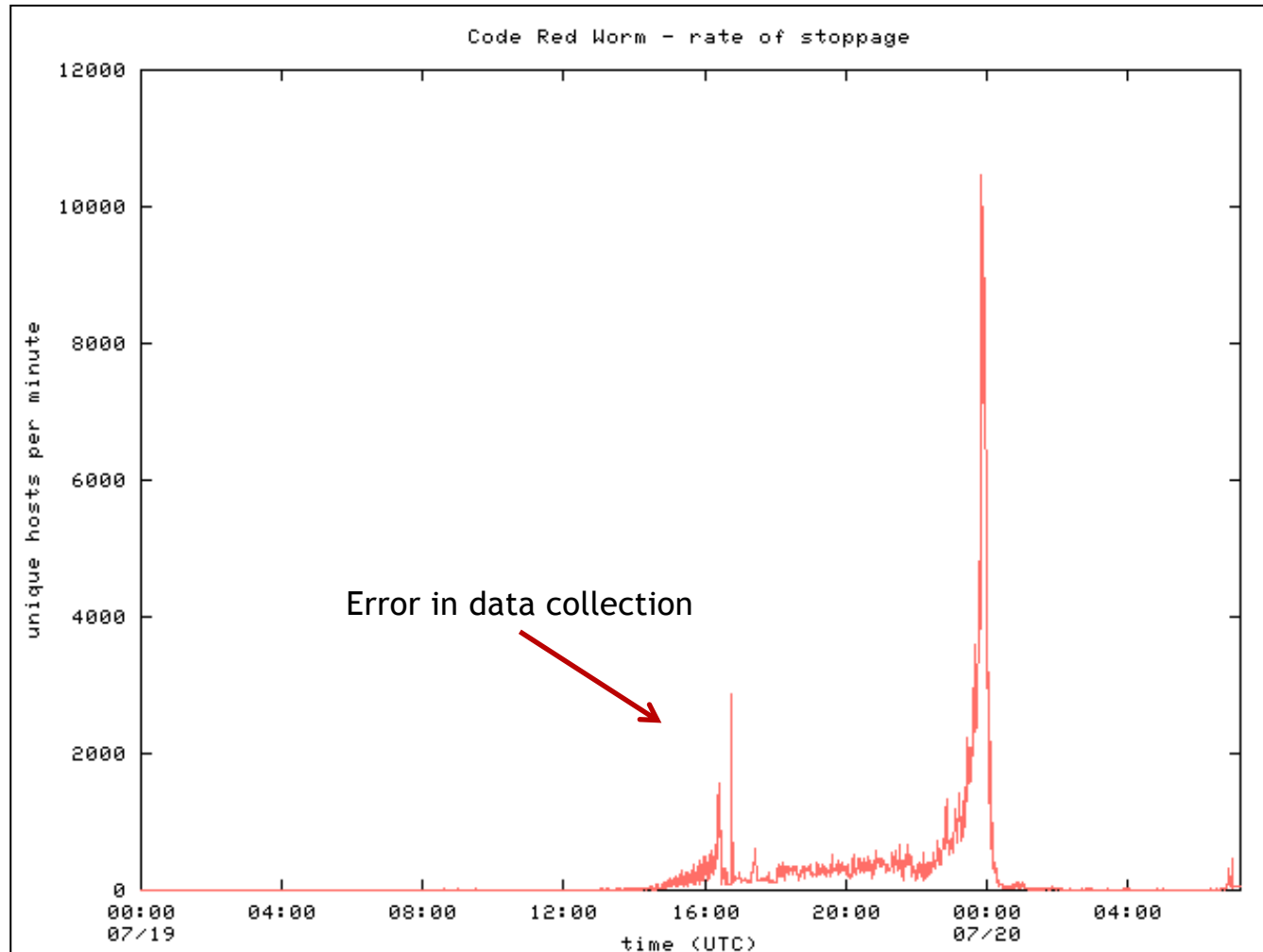
**Recall:** Code Red stopped infecting on the 20<sup>th</sup> of each month

---

**Question:** Why was the spread rate of Code Red so much higher than that of the Morris worm?



Throughout the day, many hosts were patched or firewalled to help prevent the spread of Code Red



We got lucky here...

# Characterizing the Attack

## *Top 10 Countries*

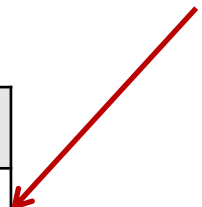
<i>Country</i>	<i>Hosts Infected</i>	<i>Percentage</i>
United States	157694	43.91
Korea	37948	10.57
China	18141	5.05
Taiwan	15124	4.21
Canada	12469	3.47
United Kingdom	11918	3.32
Germany	11762	3.28
Australia	8587	2.39
Japan	8282	2.31
Netherlands	7771	2.16

# Characterizing the Attack


*Top 10 TLDs*

<i>TLD</i>	<i>Hosts Infected</i>	<i>Percentage</i>
Unknown	169584	47.22
net	67486	18.79
com	51740	14.41
edu	8495	2.37
tw	7150	1.99
jp	4770	1.33
ca	4003	1.11
it	3076	0.86
fr	2677	0.75
nl	2633	0.73

No reverse lookup entry



Consistent with the overall  
representation of these  
TLDs on the Internet



# Characterizing the Attack

## *Top 10 Domains*

<i>Domain</i>	<i>Hosts Infected</i>	<i>Percentage</i>
Unknown	169584	47.22
home.com	10610	2.95
rr.com	5862	1.63
t-dialin.net	5514	1.54
pacbell.net	3937	1.10
uu.net	3653	1.02
aol.com	3595	1.00
hinet.net	3491	0.97
net.tw	3401	0.95
edu.tw	2942	0.82

**Note:** Home and small business ISPs played a huge role in the spread of Code Red!

# Outcomes of Code Red

**Main point:** We got lucky 😊

- Code Red was a fairly benign worm
- The automatic cut-off date eased the disinfection process
- The worm relied on a flawed DDoS attack strategy

Code Red taught us a number of important lessons

- Home users play a big role in worm propagation
- Homogeneity makes the Internet susceptible to widespread attack
- Even a worm that randomly guesses IP addresses can spread at an alarming rate
- A “release and patch” mentality is detrimental

---

**Question:** If a random scanning worm can infect over 359,000 hosts in 13 hours, what could a more directed worm do?



# Researchers have predicted that the spread of flash worms could happen too fast to stop

Random probing slows worms down

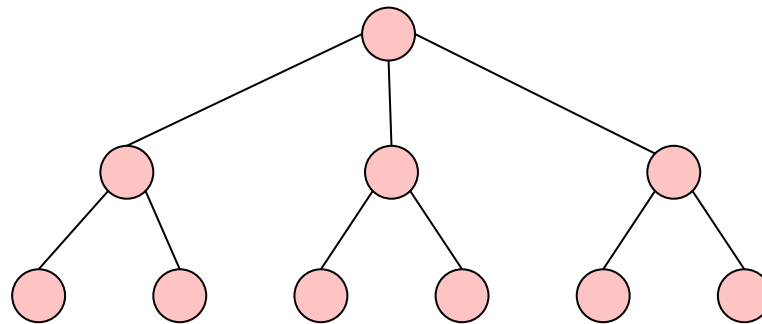
- Attempts to attack non-existent hosts
- Infecting hosts with minimal ability to infect others
- Hosts can be infected multiple times

**Flash worms** seek to spread as quickly as possible!

*Phase 1:* Find Vulnerable Hosts



*Phase 2:* Build an optimized attack tree



*Phase 3:* Infect!



So, **how fast** could a flash worm spread in the wild?

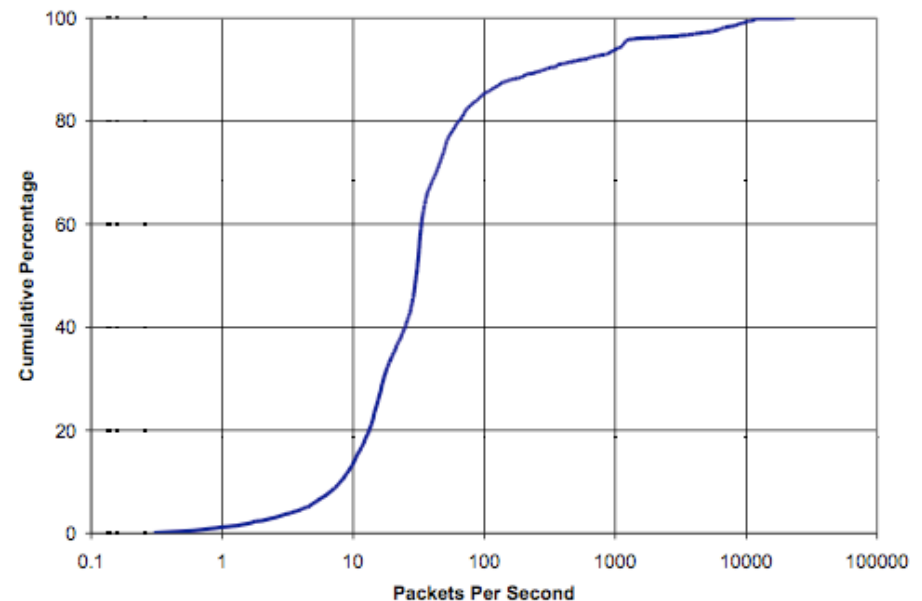
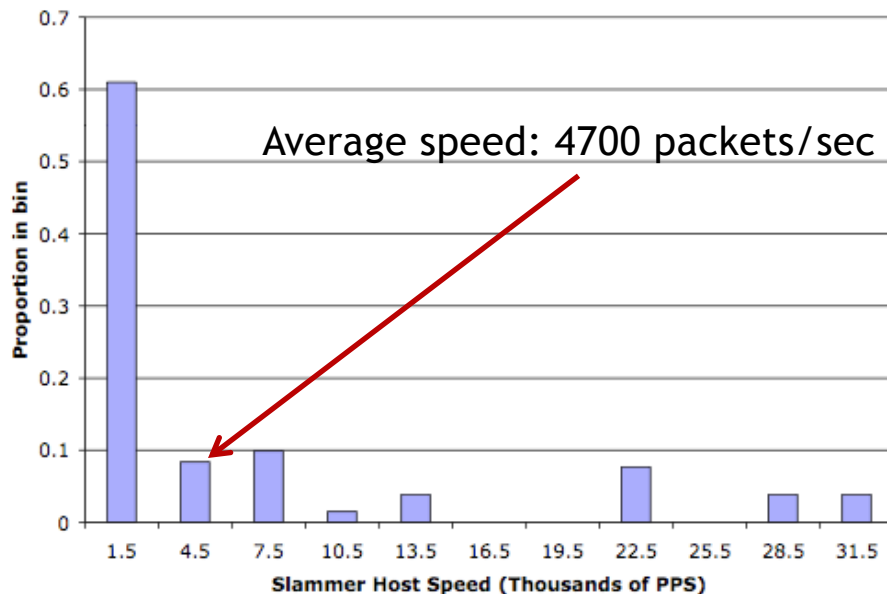
Stuart Staniford, David Moore, Vern Paxson, and Nicholas Weaver, "The Top Speed of Flash Worms," Proceedings of the ACM Workshop on Rapid Malcode (WORM), Oct. 2004.

# Predicting a worm means that we first need to characterize it

How small could a flash worm be?

- The **Slammer worm** used a single 404-byte UDP packet!

How fast could a flash worm propagate?



60% of nodes infected by Witty sent between 11 and 60 1090-byte packets/sec. This would be between 29.67 and 161.88 Slammer-sized packets/sec.

# Flash worms use an optimized distribution tree

Average Internet latency distribution is 103ms.

- Sending Slammer sized packets at 2700pps, 227 packets can be sent before the first infection
- This motivates a wide and shallow infection tree

Time to infection can be modeled as follows:

The diagram shows the equation for the time to infection,  $t_I$ , with various components annotated by red arrows and brackets. The equation is:

$$t_I = \frac{N(W + 4A)}{(A + 1)B} + \frac{AW}{b} + 2L$$

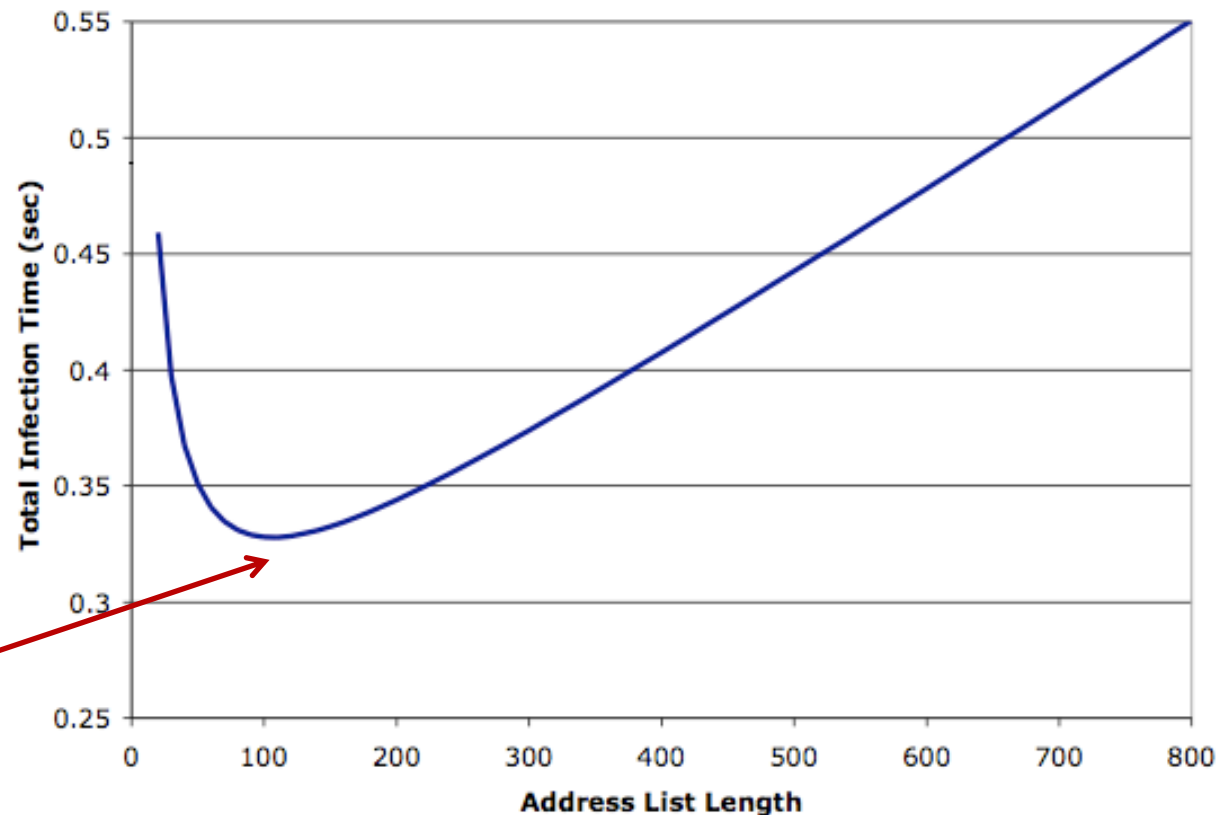
Annotations and their corresponding parts of the equation:

- Number of hosts to infect**: Points to  $N$ .
- Size of worm packet**: Points to  $W$ .
- Number of addresses sent to each node**: Points to  $4A$ .
- Latency**: Points to  $L$ .
- Bandwidth**: Points to  $b$ .
- Time to infect first level in tree**: A bracket under  $\frac{N(W + 4A)}{(A + 1)B}$  points to this label.
- Time to infect second level in tree**: A bracket under  $\frac{AW}{b}$  points to this label.
- Parallelized latency**: A bracket under  $2L$  points to this label.

# What is the optimal number of addresses to send to each 2<sup>nd</sup> level host?

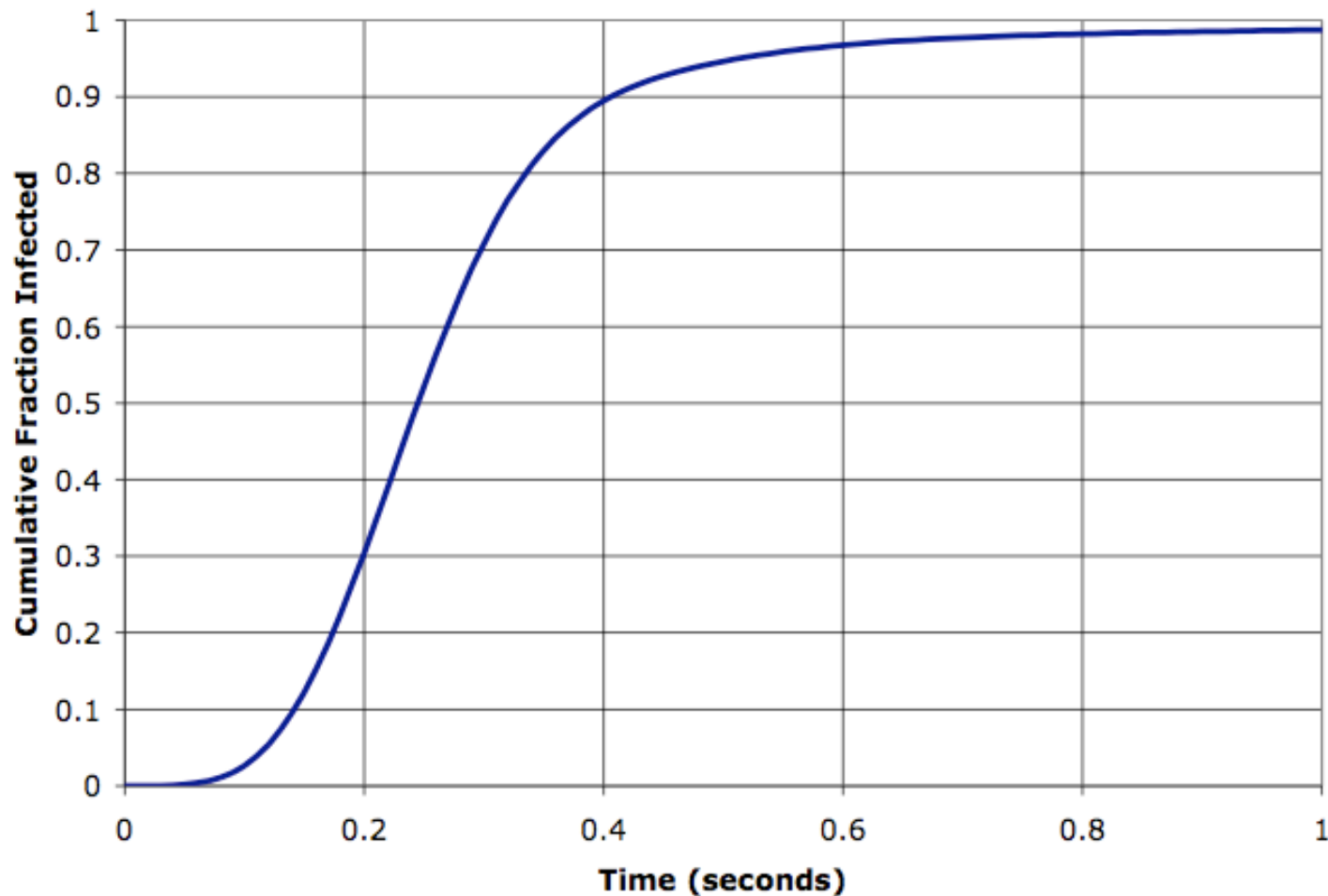
## Assumptions:

- $N = 1,000,000$  hosts to infect
- $W = 404$  bytes
- Initial link can send 240,000 Slammer-sized packets/sec
  - 75% of a 1 Gbps link
  - $B = .75$  Gbps
- $L = 103$  ms
- $b = 1$  Mbps

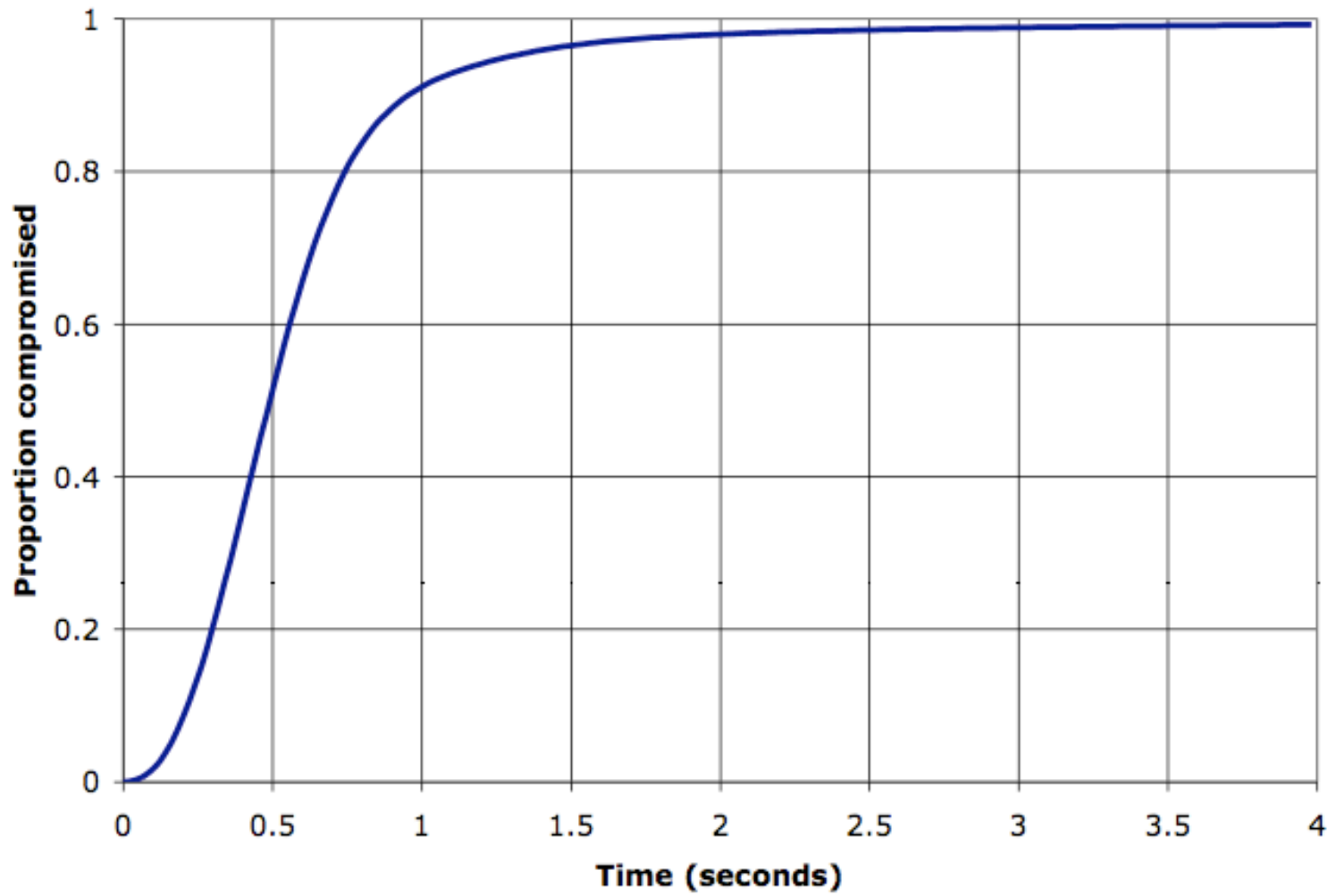


**Optimum:** 107 addresses to each 2<sup>nd</sup> level host

A UDP flash worm could infect 95% of 1 million susceptible hosts in 510ms!



A TCP flash worm could infect 95% of 1 million susceptible hosts in 3.3 seconds!



# What about more sophisticated, targeted attacks?

## Stuxnet

- Detected in 2010, developed starting in 2005
- Now believed to be developed by US and Israel
- Spread via Windows, targeted Siemens industrial systems
- Reportedly ruined 20% of Iran's nuclear centrifuges

## Flame

- Discovered in 2012, in the wild since at least 2007
- Now known developed by NSA, Israel, and GCHQ for cyber espionage
- Mutated and self-destructed to evade detection
- Exploited weaknesses in MD5 to counterfeit certificate
- Records audio, screenshots, key presses, network traffic, nearby bluetooth devices' contacts, etc.
- Reportedly infected at least 1,000 government, education, and private computers

# Summary

Malicious code has been around for a very long time

In the early days, computer viruses and Trojan horses moved at the speed of human-to-human interactions

Worms spread much faster by leveraging constant node connectivity

- Over the years, the propagation techniques used by worms have not changed
- More aggressive propagation mechanisms allow newer worms to spread faster

Flash worms are quite scary, but sensitive to minor problems in the network

- Excellent “worse case” for analyzing worm defenses