Information Security 12

Software Security

Chapter 3 in Security in Computing,
Charles P. Pfleeger, Shari Lawrence Pfleeger,
Pearson Edition

Why Software?

- Why is software as important to security as crypto, access control and protocols?
- Virtually all of information security is implemented in software
- If your software is subject to attack, your security is broken
  - Regardless of strength of crypto, access control or protocols
- Software is a poor foundation for security

What does it mean?

- “secure” program: means different things to different people
- is it secure if ?
  - takes too long to break through security controls
  - runs for a long time without failure
  - it conforms to specification
  - free from all faults

Fixing Faults - Testing

- which is better:
  - finding and fixing 20 faults in a module?
  - finding and fixing 100 faults ' ' ' ?

Fixing Faults

- which is better:
  - finding and fixing 20 faults in a module?
  - finding and fixing 100 faults ' ' ' ?
- finding 100 could mean
  - you have better testing methods
  - OR
    - code is really bad
    - 100 were just the tip of the iceberg
  - software testing literature:
    - finding many errors early → probably find many more

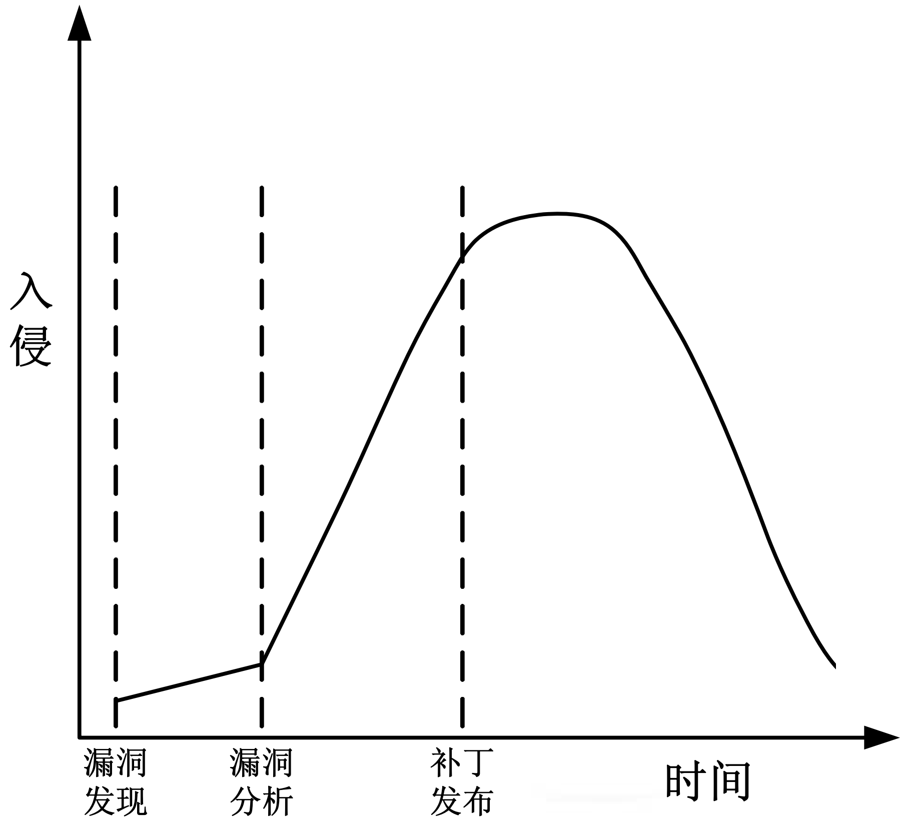
Fixing Faults: penetrate and patch

- think of security *after* program has been broken
- release a patch
- why is this bad?

Fixing Faults: penetrate and patch

- think of security *after* program has been broken
- release a patch
- why is this bad?
- eg.

  Unicode,MS00-057



Fixing Faults: penetrate and patch

- why is this bad?
  - product was broken in the first place
  - developers can only fix problems that they know about
  - patches often only fix symptom. they're not cure
  - people don't bother applying the patches
  - patches can have holes
  - patches tell the bad guys where the problems are
  - might affect program performance or limit functionality
  - more expensive than making it secure from the beginning

Software Issues

"Normal" users

- Find bugs and flaws by accident
- Hate bad software…
- …but must learn to live with it
- Must make bad software work

Attackers

- Actively look for bugs and flaws
- Like bad software…
- …and try to make it misbehave
- Attack systems thru bad software

Complexity

- "*Complexity is the enemy of security*", Paul Kocher, Cryptography Research, Inc.

system	Lines of code (LOC)
Netscape	17,000,000
Space shuttle	10,000,000
Linux	1,500,000
Windows XP	40,000,000
Boeing 777	7,000,000

- A new car contains more *LOC* than was required to land the Apollo astronauts on the moon

Lines of Code and Bugs

- Conservative estimate: 5 bugs/1000 LOC
- **Do the math**
  - Typical computer: 3,000 exe's of 100K each
  - Conservative estimate of 50 bugs/exe
  - About 150k bugs per computer
  - 30,000 node network has 4.5 billion bugs
  - Suppose that only 10% of bugs security-critical and only 10% of those remotely exploitable
  - Then "only" 4.5 million critical security flaws!

Complete Program Security

- Can we make programs completely secure?
  - Not easy
- Why?
  - Software testing:
    - makes sure that code does what it's supposed to do
  - for security: must also verify that it doesn't do anything it isn't supposed to do. *much harder*
  - programming techniques often change more quickly than security techniques

Software Security Topics

- Program flaws (unintentional)
  - Buffer overflow
  - Incomplete mediation
  - Race conditions
- Malicious software (intentional)
  - Viruses
  - Worms
  - Other breeds of malware

Program Flaws

- An **error** is a programming mistake
  - To err is human
- An error may lead to incorrect state: **fault**
  - A fault is internal to the program
- A fault may lead to a **failure**, where a system departs from its expected behavior
  - A failure is externally observable

$$\begin{tikzcd} \text{error} \arrow[r] & \text{fault} \arrow[r] & \text{failure} \end{tikzcd}$$

Secure Software

- In software engineering, try to insure that a program does what is intended
- Secure software engineering requires that the software **does what is intended…**
- **…and nothing more**
- Absolutely secure software is impossible
  - Absolute security is almost never possible!
- How can we manage the risks?

Program Flaws

- Program flaws are unintentional
  - But still create security risks
- We'll consider 3 types of flaws
  - Buffer overflow (smashing the stack)
  - Incomplete mediation
  - Race conditions
- Many other flaws can occur
- These are most common

Buffer Overflow-the first enemy

- Cause by bad programming practices
- Most common security vulnerability
  - 9 of 13 CERT advisories from 1998
  - at least half of 1999 CERT advisories (8/17)
  - 18 of 28 CERT advisories from 2003
- Most of the exploits based on buffer overflows aim at forcing the execution of malicious code.
- Problems
  - Access an array without boundary checking
  - String specification in C/C++ (end with NULL)

Typical Attack Scenario

- Users enter data into a Web form
- Web form is sent to server
- Server writes data to buffer, without checking length of input data
- Data overflows from buffer
- Sometimes, overflow can enable an attack
- Web form attack could be carried out by anyone with an Internet connection

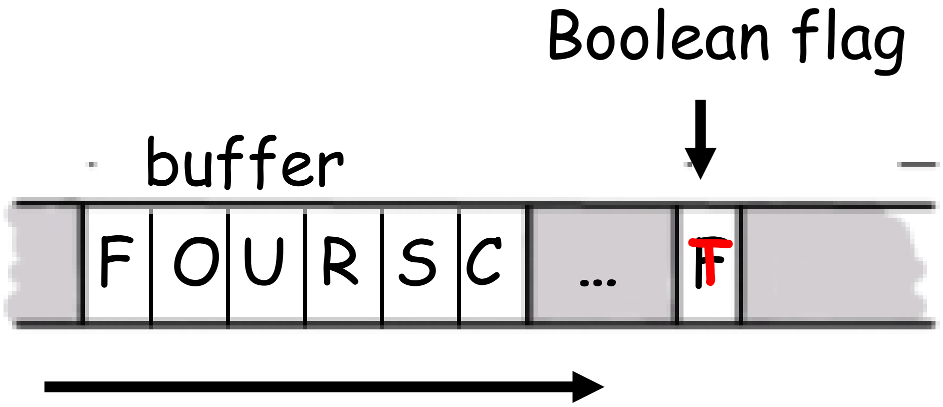
Buffer Overflow

```
int main(){
   int buffer[10];
   buffer[20] = 37;}
```

- **Q:** What happens when this is executed?
- **A:** Depending on what resides in memory at location “buffer[20]”
  - Might overwrite **user** data or code
  - Might overwrite **system** data or code

Simple Buffer Overflow

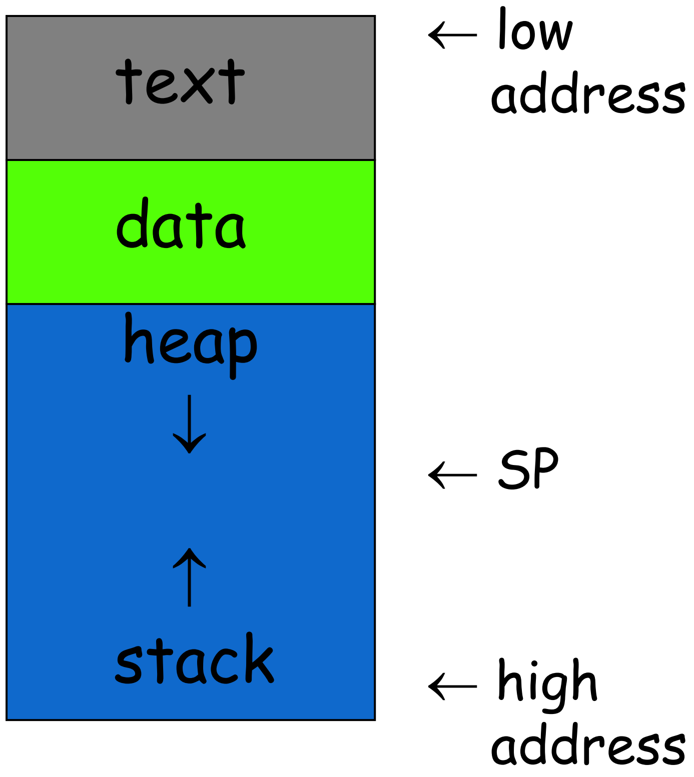
- Consider boolean flag for authentication
- Buffer overflow could overwrite flag allowing anyone to authenticate!



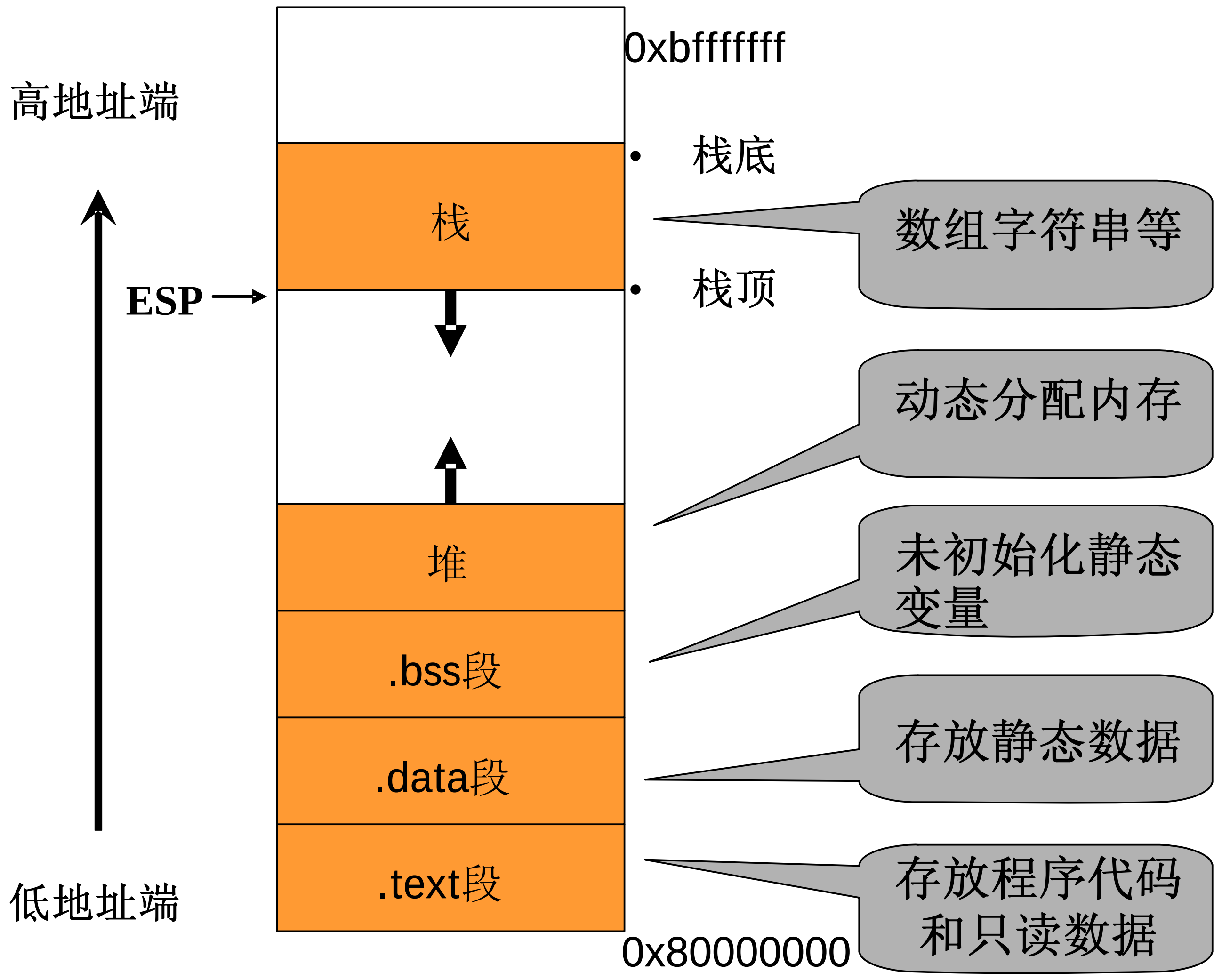
- In some cases, attacker need not be so lucky as to have overflow overwrite flag

Memory Organization

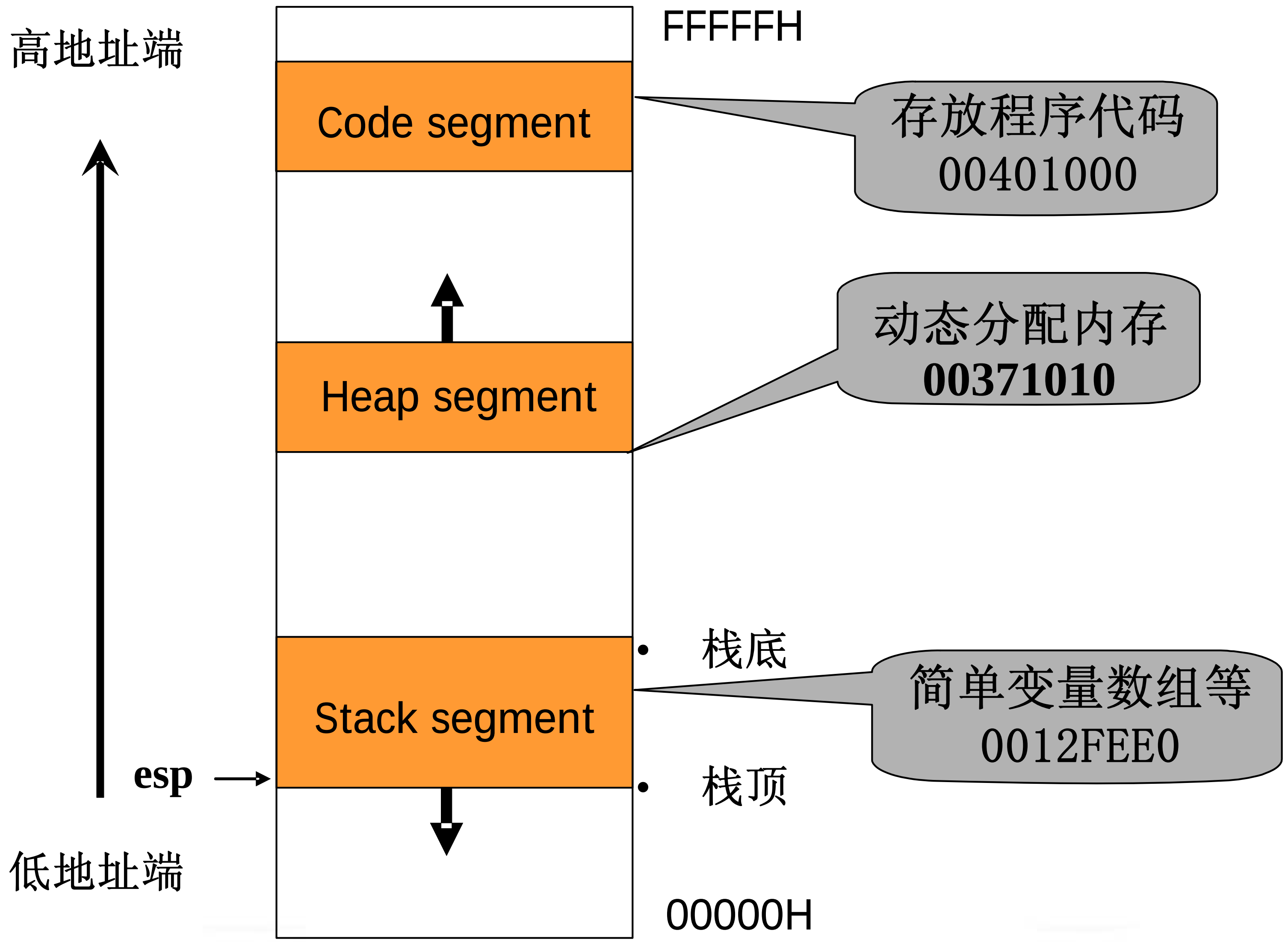
- **Text** == code
- **Data** == static variables
- **Heap** == dynamic data
- **Stack** == "scratch paper"
  - Dynamic local variables
  - Parameters to functions
  - Return address



程序在内存中的映射（linux）

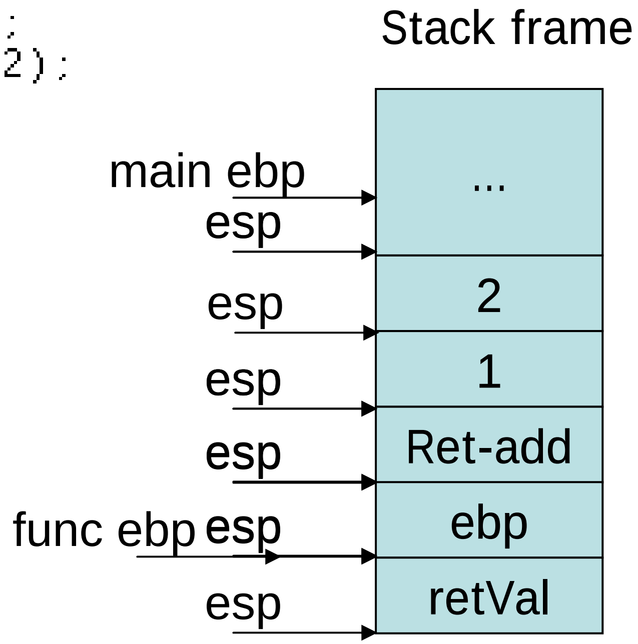


程序在内存中的映射(Win32)



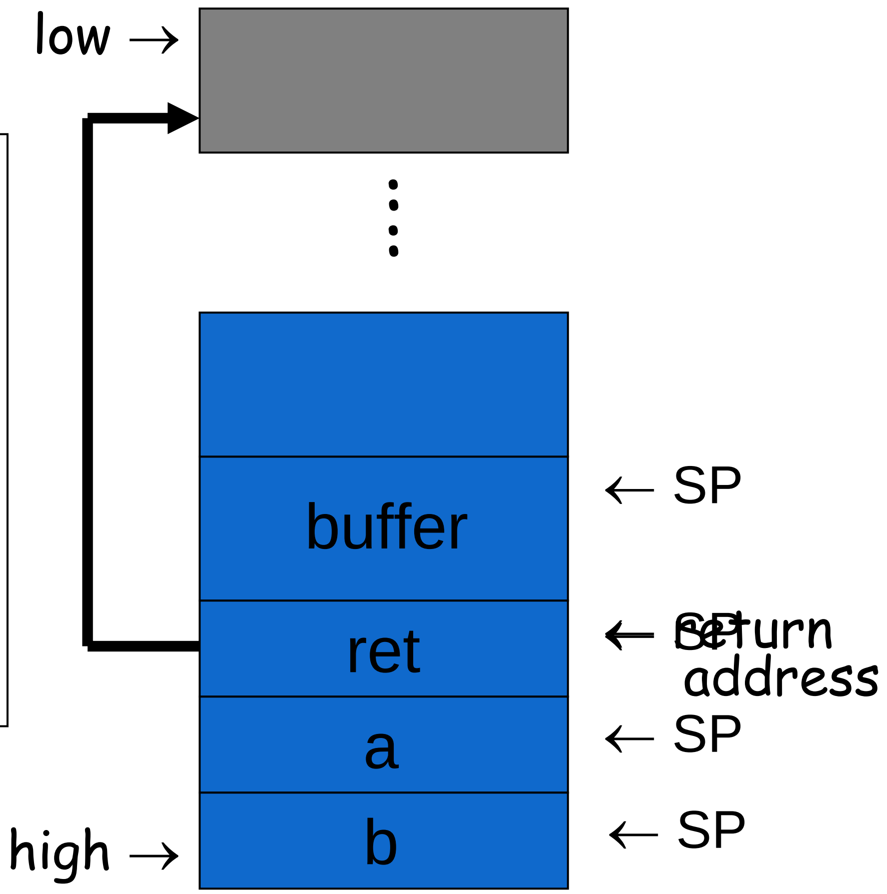
函数调用过程示例

```c
int func(int a, int b){
  int retVal = a + b;
  printf("b: 0x%08x\n",&b);
  printf("a: 0x%08x\n",&a);
  printf("ret addr here: 0x%08x\n",&a-1);
  printf("stored ebp here: 0x%08x\n",&a-2);
  printf("retVal: 0x%08x\n\n",&retVal);
  return retVal;
}
int main(int argc, char* argv[])
{
    int result = func(1, 2);
    return 0;
}
```



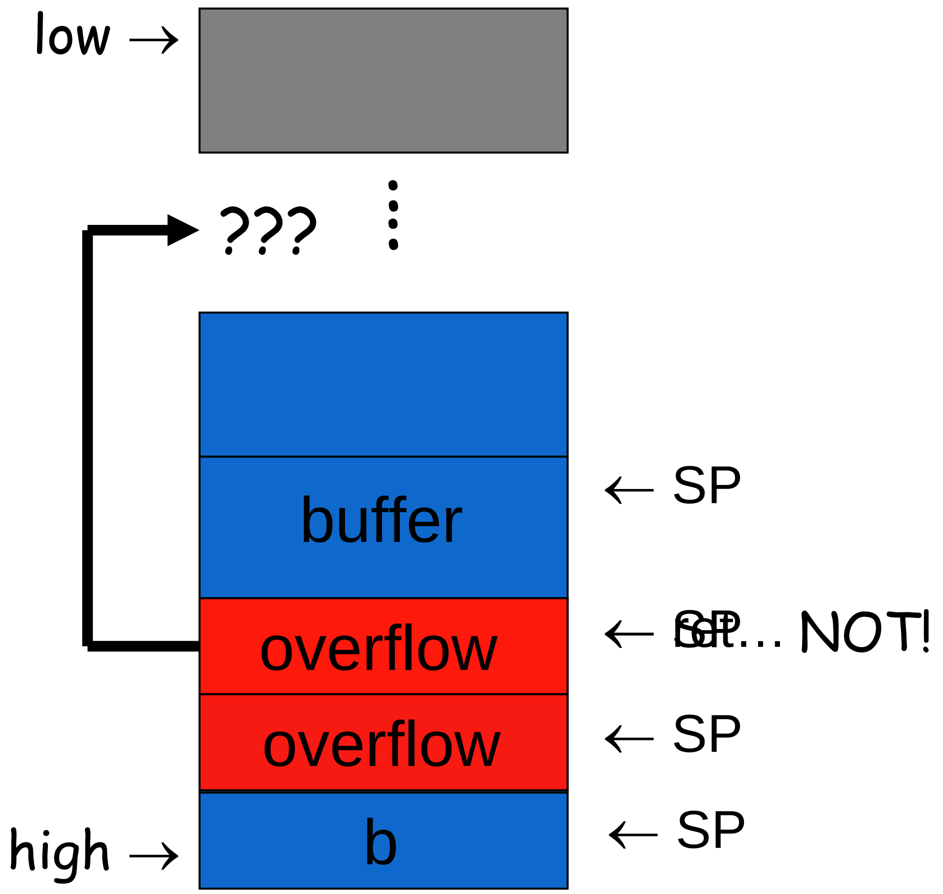
Simplified Stack Example

```
void func(int a, int b){
    char buffer[10];
}
void main(){
    func(1, 2);
}
```



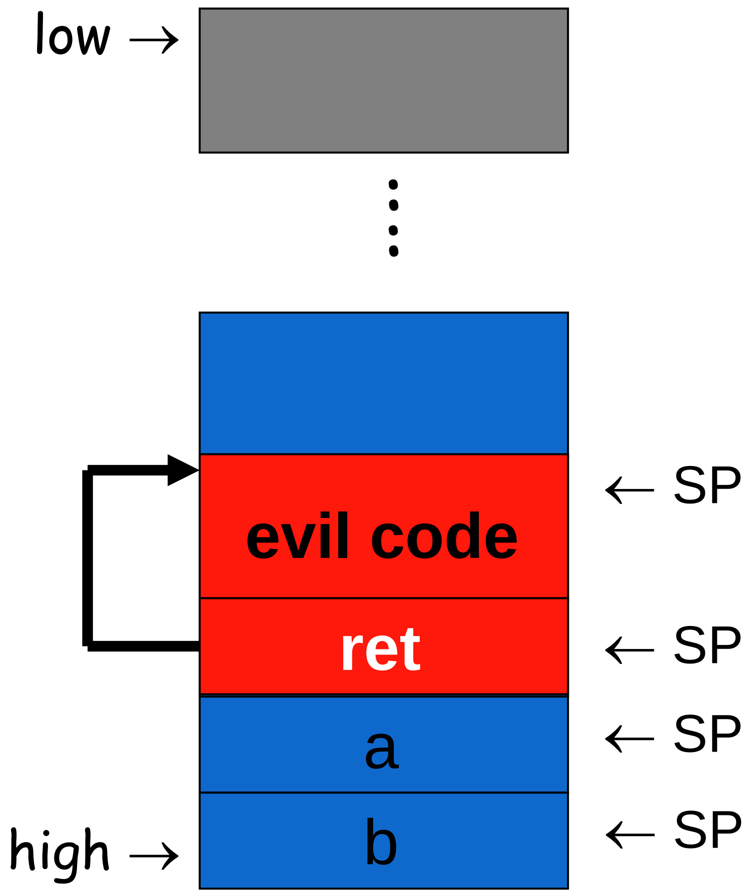
Smashing the Stack

- What happens if buffer overflows?
- Program "returns" to wrong location
- A crash is likely



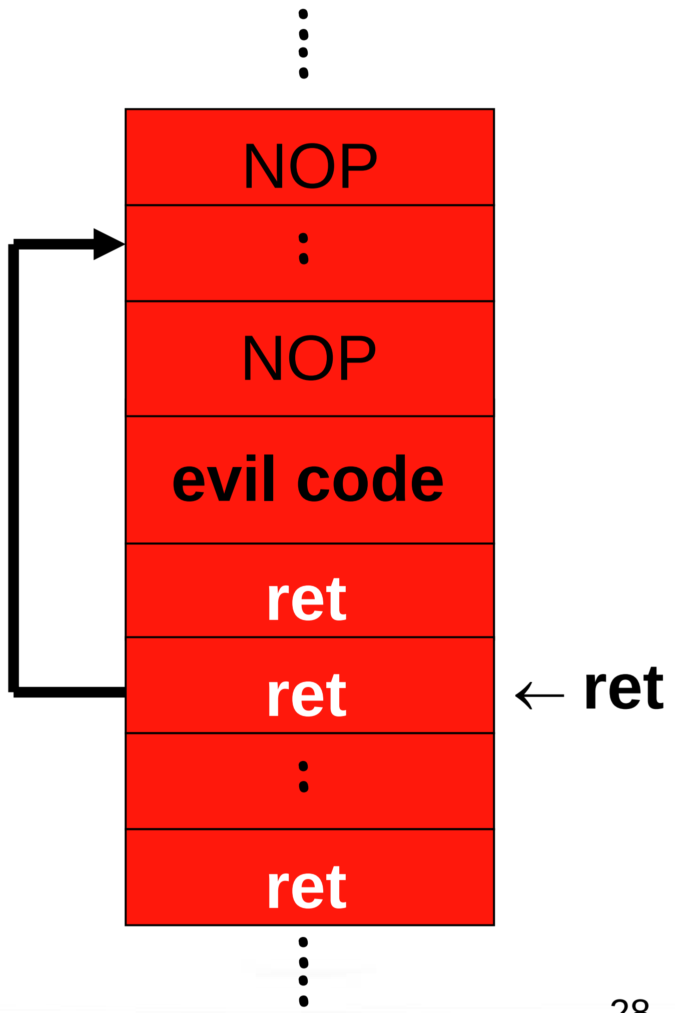
Smashing the Stack

- Trudy has a better idea…
- **Code injection**
- Trudy can run code of her choosing!



Smashing the Stack

- Trudy may not know
  - Address of evil code
  - Location of **ret** on stack
- Solutions
  - Precede evil code with NOP "landing pad"
  - Insert lots of new **ret**

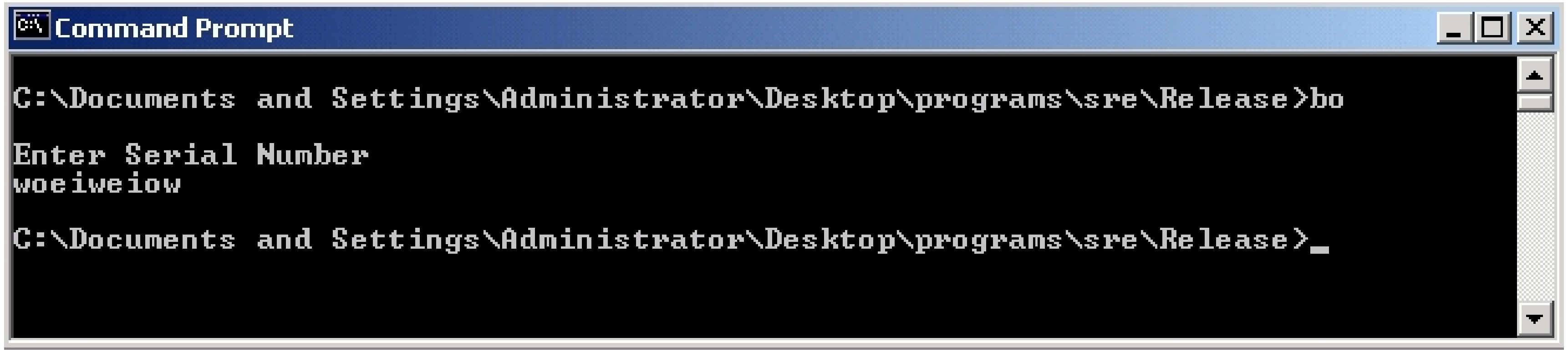


Stack Smashing Summary

- A buffer overflow must exist in the code
- Not all buffer overflows are exploitable
  - Things must line up just right
- If exploitable, attacker can **inject code**
- Trial and error likely required
  - Lots of help available online
  - Smashing the Stack for Fun and Profit, Aleph One
- Also heap overflow, integer overflow, etc.
- Stack smashing is "attack of the decade"

Stack Smashing Example

- Program asks for a serial number that the attacker does not know
- Attacker does **not** have source code
- Attacker does have the executable (exe)

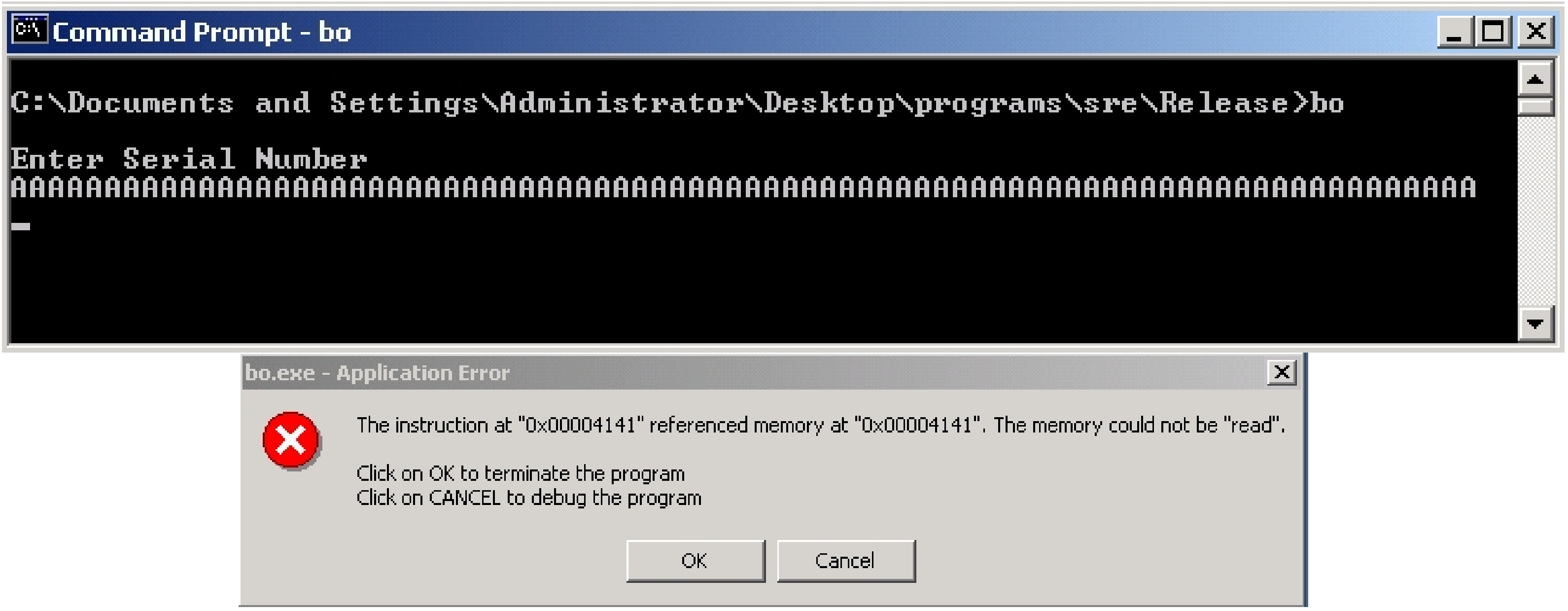


```
C:\Documents and Settings\Administrator\Desktop\programs\sre\Release>bo

Enter Serial Number
woeiweiow

C:\Documents and Settings\Administrator\Desktop\programs\sre\Release>_
```

- Program quits on incorrect serial number

Example

- By trial and error, attacker discovers an apparent buffer overflow



```
C:\Documents and Settings\Administrator\Desktop\programs\sre\Release>bo

Enter Serial Number
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

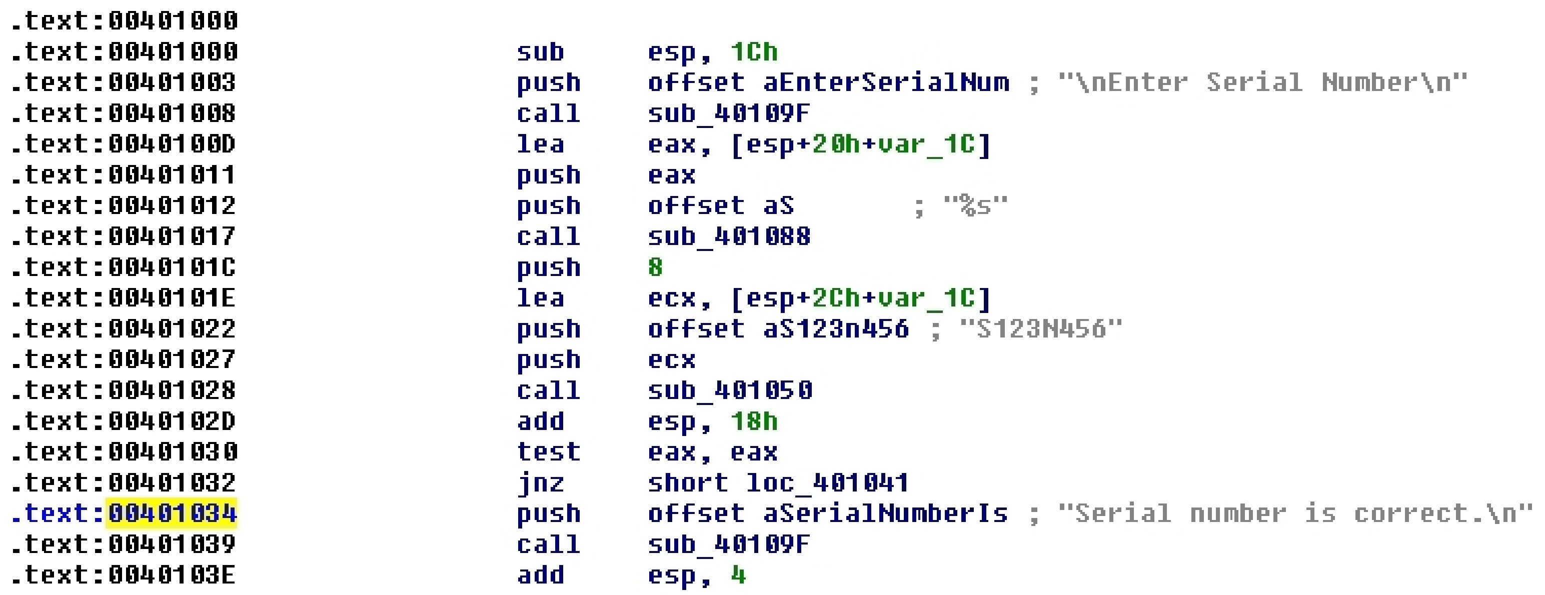
The instruction at "0x00004141" referenced memory at "0x00004141". The memory could not be "read".

Click on OK to terminate the program
Click on CANCEL to debug the program

- Note that 0x41 is "A"
- Looks like **ret** overwritten by 2 bytes!

Example

- Next, disassemble bo.exe to find

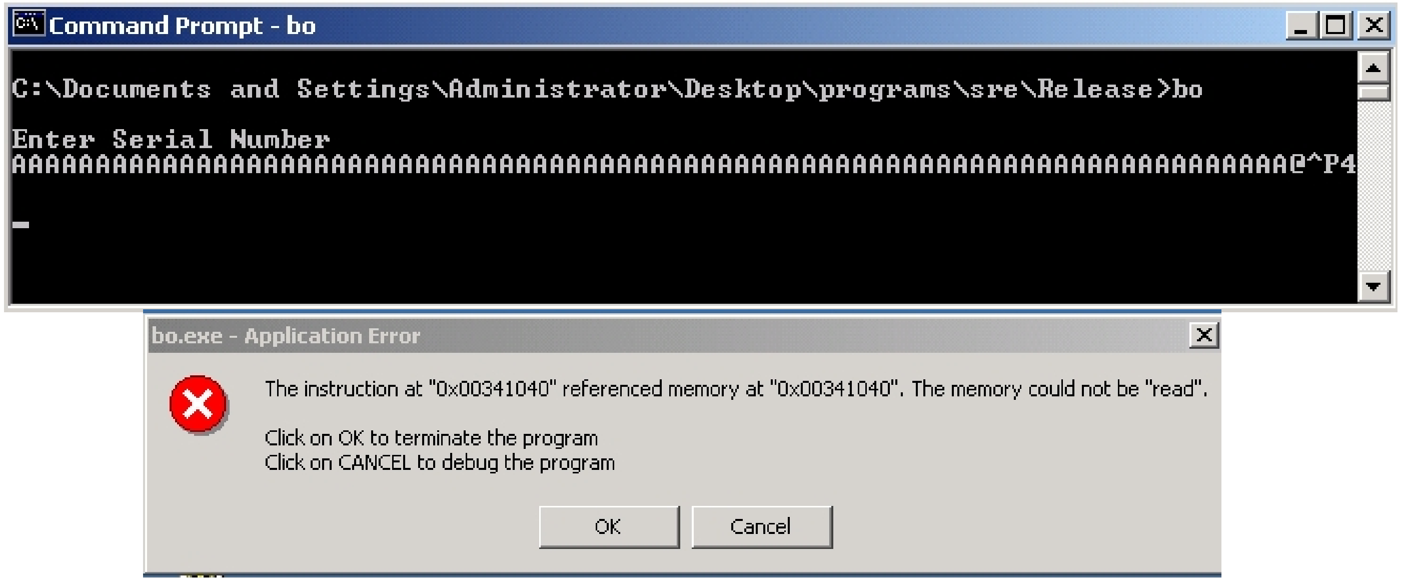


```
.text:00401000
.text:00401000                 sub     esp, 1Ch
.text:00401003                 push    offset aEnterSerialNum ; "\nEnter Serial Number\n"
.text:00401008                 call    sub_40109F
.text:0040100D                 lea     eax, [esp+20h+var_1C]
.text:00401011                 push    eax
.text:00401012                 push    offset aS       ; "%s"
.text:00401017                 call    sub_401088
.text:0040101C                 push    8
.text:0040101E                 lea     ecx, [esp+2Ch+var_1C]
.text:00401022                 push    offset aS123n456 ; "S123N456"
.text:00401027                 push    ecx
.text:00401028                 call    sub_401050
.text:0040102D                 add     esp, 18h
.text:00401030                 test    eax, eax
.text:00401032                 jnz     short loc_401041
.text:00401034                 push    offset aSerialNumberIs ; "Serial number is correct.\n"
.text:00401039                 call    sub_40109F
.text:0040103E                 add     esp, 4
```

- The goal is to exploit buffer overflow to jump to address 0x401034

Example

- Find that 0x401034 is “@^P4” in ASCII



```
C:\Documents and Settings\Administrator\Desktop\programs\sre\Release>bo
Enter Serial Number
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA@^P4
_
```

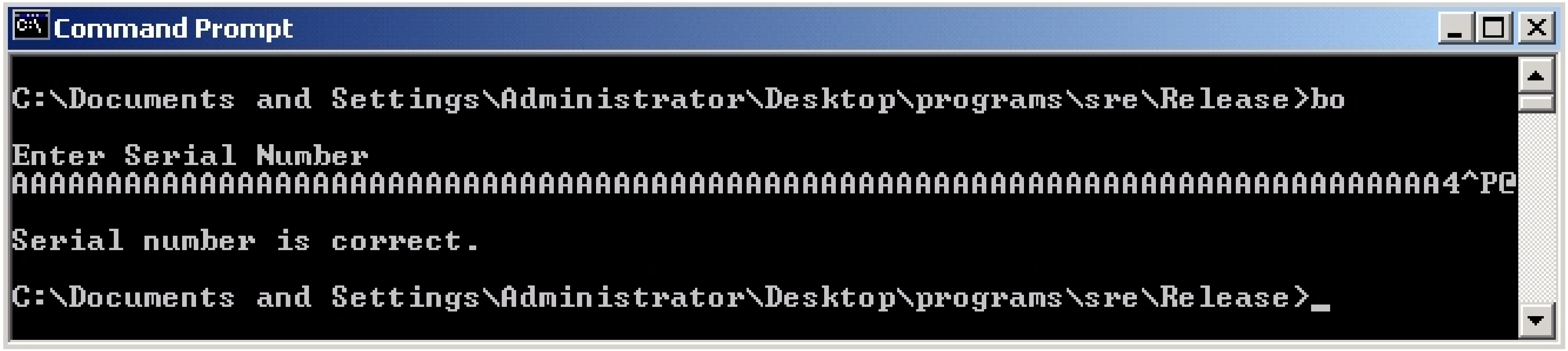
The instruction at "0x00341040" referenced memory at "0x00341040". The memory could not be "read".

Click on OK to terminate the program
Click on CANCEL to debug the program

- Byte order is reversed? Why?
- X86 processors are “little-endian“

Example

- Reverse the byte order to “4^P@” and…



```
C:\Documents and Settings\Administrator\Desktop\programs\sre\Release>bo

Enter Serial Number
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA4^P@

Serial number is correct.

C:\Documents and Settings\Administrator\Desktop\programs\sre\Release>_
```

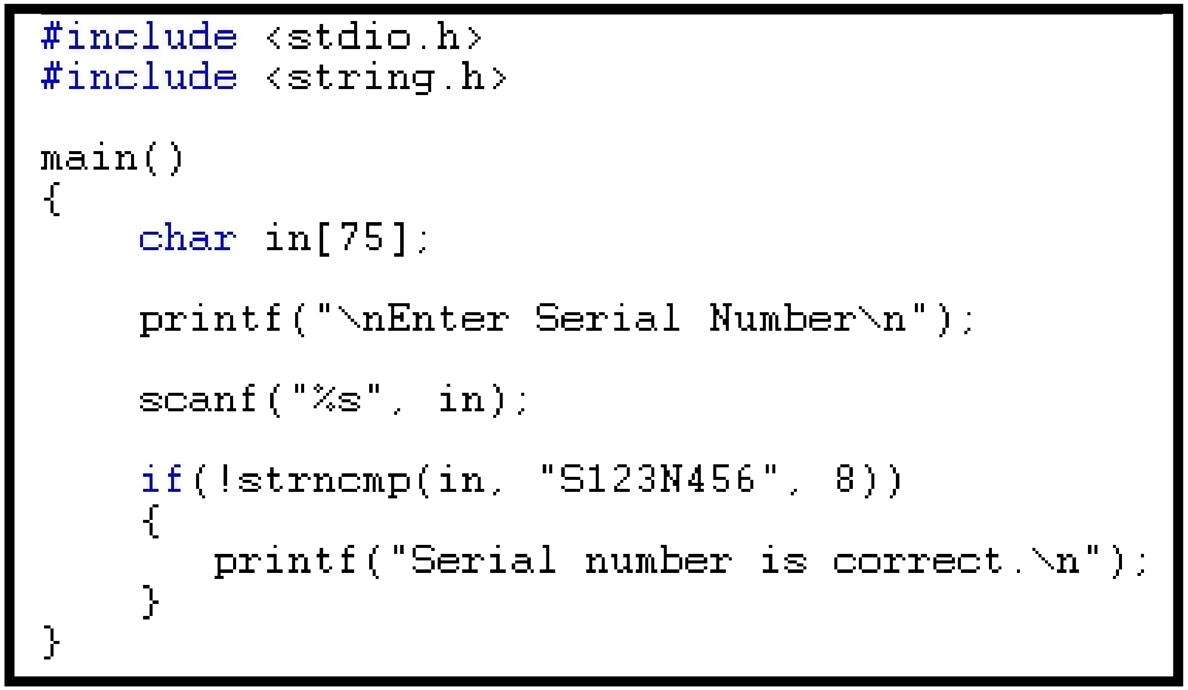
- Success! We’ve bypassed serial number check by exploiting a buffer overflow
- Overwrote the return address on the stack

Example

- Attacker did not require access to the source code
- Only tool used was a disassembler to determine address to jump to
- Can find address by trial and error
  - Necessary if attacker does not have exe
  - For example, a remote attack

Example

- Source code of the buffer overflow
- Flaw easily found by attacker
- **Even without the source code!**



```
#include <stdio.h>
#include <string.h>

main()
{
    char in[75];

    printf("\nEnter Serial Number\n");

    scanf("%s", in);

    if(!strncmp(in, "S123N456", 8))
    {
       printf("Serial number is correct.\n");
    }
}
```

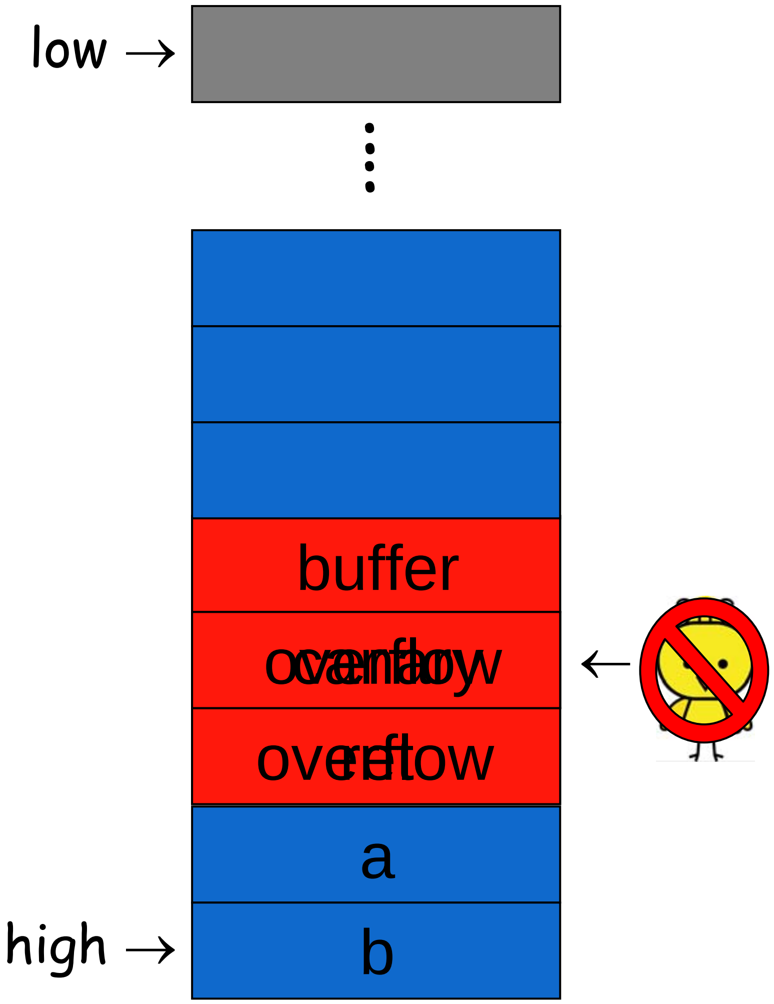
Stack Smashing Prevention

- 1st choice: employ **non-executable stack**
  - "No execute" **NX bit** (if available)
  - Seems like the logical thing to do, but some real code executes on the stack (Java does this)
- 2nd choice: use **safe languages** (Java, C#)
- 3rd choice: use **safer C functions**
  - For unsafe functions, there are safer versions
  - For example, strncpy instead of strcpy
- 4th choice: **Static source code analysis**.

Stack Smashing Prevention

- **Canary**
  - Run-time stack check
  - Push canary onto stack
  - Canary value:
    - Constant 0x000aff0d
    - Or value depends on **ret**
    - random number

- ***VC++ with /GS compiler flag***



Buffer Overflow

- The “attack of the decade” for 90’s
- Will be the attack of the decade for 00’s
- Can be prevented
  - Use safe languages/safe functions
  - Educate developers, use tools, etc.
- Buffer overflows will exist for a long time
  - Legacy code
  - Bad software development

Software Security Topics

- Program flaws (unintentional)
  - Buffer overflow
  - Incomplete mediation
  - Race conditions
- Malicious software (intentional)
  - Viruses
  - Worms
  - Other breeds of malware

Input Validation

- Consider: `strcpy(buffer, argv[1])`
- A buffer overflow occurs if

  `len(buffer) < len(argv[1])`
- Software must **validate** the input by checking the length of `argv[1]`
- Failure to do so is an example of a more general problem: **incomplete mediation**

Input Validation

- Consider web form data
- Suppose input is validated on client
- For example, the following is valid

```
http://www.things.com/orders/final&custID=112&num
  =55A&qty=20&price=10&shipping=5&total=205
```

- Suppose input is not checked on server
  - Why bother since input checked on client?
  - Then attacker could send http message

```
http://www.things.com/orders/final&custID=112&num
  =55A&qty=20&price=1&shipping=5&total=25
```

Incomplete Mediation

- Linux kernel
  - Research has revealed many buffer overflows
  - Many of these are due to incomplete mediation
- Linux kernel is "good" software since
  - Open-source
  - Kernel — written by coding gurus
- Tools exist to help find such problems
  - But incomplete mediation errors can be subtle
  - And tools useful to attackers too!

Race Conditions

Race Condition

- Security processes should be **atomic**
  - Occur "all at once"
- Race conditions can arise when security-critical process occurs in stages
- Attacker makes change between stages
  - Often, between stage that gives authorization, but before stage that transfers ownership

Race condition

- Necessary properties for a race condition
  - Concurrency property
    - At least two control flows executing concurrently
    - If not controlled can lead to nondeterministic behavior
  - Shared object property
    - The concurrent flows must access a common shared *race object*
  - Change state property
    - Atleast one control flow must alter the state of the race object
- Software vulnerability resulting from unanticipated execution ordering of concurrent flows

Race window

- A code segment that accesses the race object in a way that opens a window of opportunity for race condition
  - Sometimes referred to as critical section
- Traditional approach
  - Ensure race windows do not overlap
    - Make them mutually exclusive
    - Language facilities – *synchronization primitives (SP)*
  - *Deadlock* is a risk related to SP
    - Denial of service

Time-of-Check-To-Time-of-Use

- Source of race conditions
  - Trusted (tightly coupled threads of execution) or untrusted control flows (separate application or process)
- ToCTToU race conditions
  - Can occur during file I/O
  - Forms a RW by first *checking* some race object and then *using* it

Example

```
int main(int argc, char *argv[]) {
        FILE *fd;
        if (access(“/some_file”, W_OK) == 0) {
                printf("access granted.\n");
                fd = fopen(“/some_file”, "wb+");
                /* write to the file */
                fclose(fd);
        }  else {
                err(1, "ERROR");
        }
        return 0;
} Figure 7-1
```

- Assume the program is running with an effective UID of root
- Present in xterm program, while logging sessions

TOCTTOU

- Following shell commands during RW

```
rm /some_file
ln /myfile /some_file
```

- Mitigation
  - Replace access() call by code that does the following
    - Drops the privilege to the real UID
    -
    - Check to ensure that the file was opened successfully

Temporary file open exploits

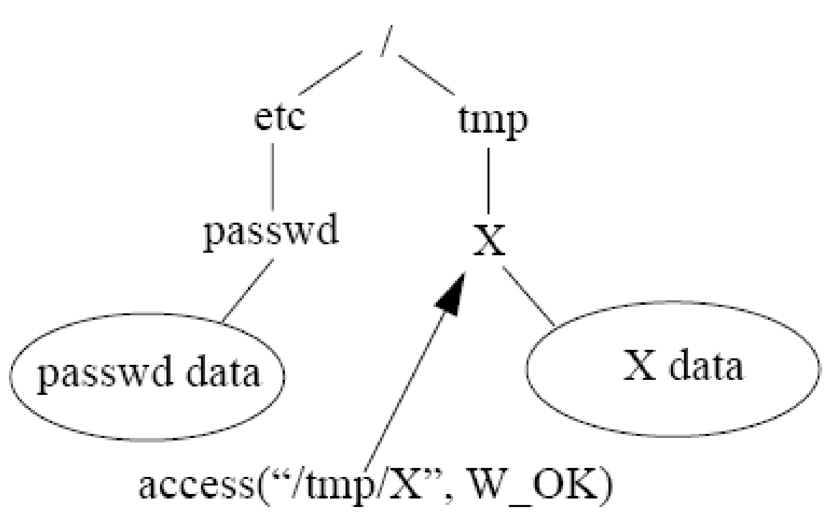
- Temporary files
  - Unique naming is difficult
  - Vulnerable when created in a directory where attacker has access
  - In unix /tmp is frequently used for temporary files
  - Simple vulnerability

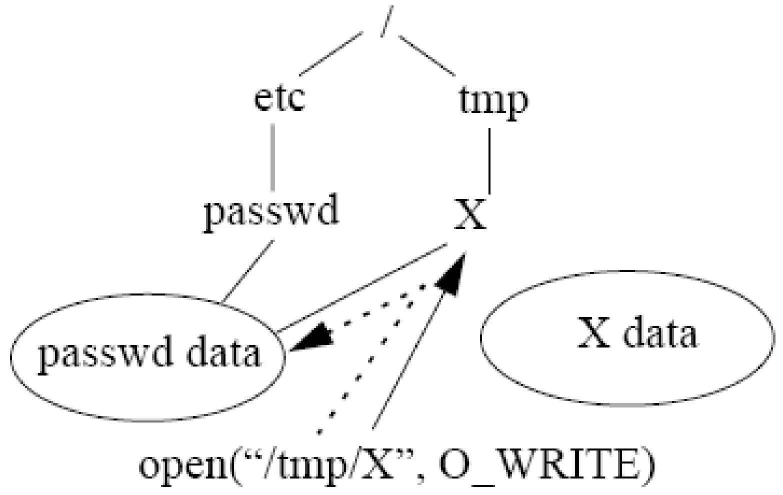
```
 int fd = open(“/tmp/some_file”,
                    O_WRONLY |
                    O_CREAT |
                    O_TRUNC,
                    0600)
```

Already exists or what if the /tmp/some_file is a symbolic link before the instruction is executed?

Solution:
add O_EXCL flag

File existence check and creation -> atomic!





Source: Bishop and Dilger's 1996 paper in Computing Systems

Race Conditions

- Race conditions are common
- Race conditions may be more prevalent than buffer overflows
- But race conditions harder to exploit
  - Buffer overflow is “low hanging fruit” today
- To prevent race conditions, make security-critical processes atomic
  - Occur all at once, not in stages
  - Not always easy to accomplish in practice

Race detection tools

- Static analysis
  - Parses software to identify race conditions
  - Warlock for C (need annotation)
  - ITS4 uses (database of vulnerabilities)
  - RacerX for control-flow sensitive interprocedural analysis
  - Flawfinder and RATS – best public domain
- Extended Static checking
  - Use theorem proving technology
- Race condition detection is NP complete
  - Hence approximate detection
  - C/C++ are difficult to analyze statically –
    - pointers and pointer arithmetic
    - Dynamic dispatch and templates in C++

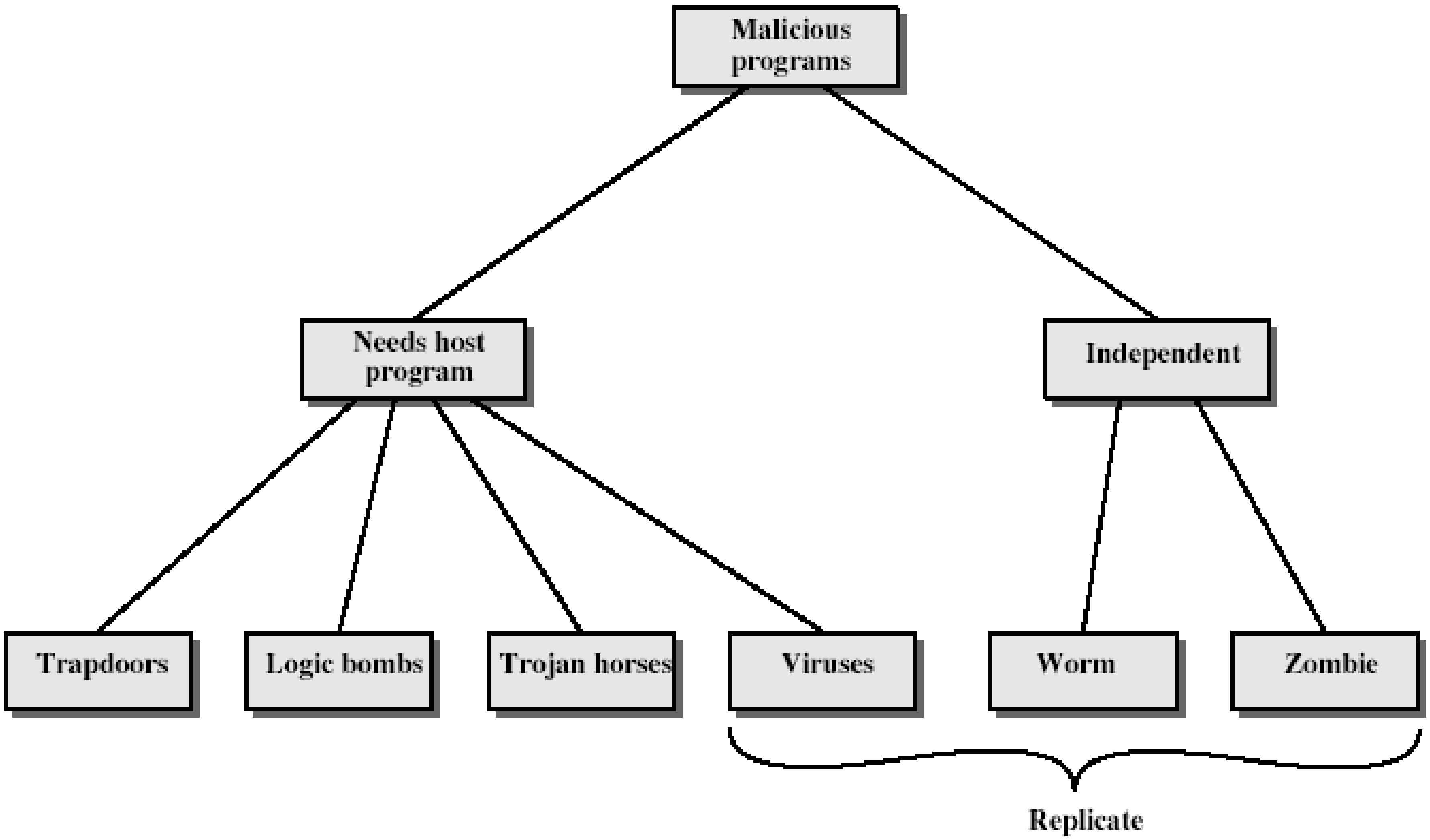
Software Security Topics

- Program flaws (unintentional)
  - Buffer overflow
  - Incomplete mediation
  - Race conditions
- **Malicious software (intentional)**
  - Viruses
  - Worms
  - Other breeds of malware

Malware

- **Malware which spread from machine to machine *without* the consent of the owners/operators/users**
  - Windows Automatic Update is (effectively) consensual
- **Many strains possible**
  - Viruses
  - Worms
  - Compromised Auto-updates
    - No user action required, very dangerous

Type of Malware (lots of overlap)



Trapdoors (Back doors)

- Secret entry point into a program
- Allows those who know access bypassing usual security procedures, e.g., authentications
- Have been commonly used by developers
- A threat when left in production programs allowing exploited by attackers
- Very hard to block in O/S
- Requires good s/w development & update

Logic Bomb

- One of oldest types of malicious software
- Code embedded in legitimate program
- Activated when specified conditions met
  - E.g., presence/absence of some file
  - Particular date/time
  - Particular user
  - Particular series of keystrokes
- When triggered typically damage system
  - Modify/delete files/disks

Trojan Horse

- Programs that appear to have one function but actually perform another.
- Modern Trojan Horse: resemble a program that the user wishes to run - usually superficially attractive
  - E.g., game, s/w upgrade etc
- When run performs some additional tasks
  - Allows attacker to indirectly gain access they do not have directly
- Often used to propagate a virus/worm or install a backdoor
- Or simply to destroy data

Zombie

- Program which secretly takes over another networked computer
- Then uses it to indirectly launch attacks
- Often used to launch distributed denial of service (DDoS) attacks
- Exploits known flaws in network systems

Malware Timeline

- Preliminary work by Cohen (early 80's)
- First Wild Viruses
- Brain virus (1986)
- Morris worm (1988)
- Destructive Virus: CIH
- Code Red (2001)
- SQL Slammer (2004)
- Future of malware?

First Wild Viruses, on Apple 1981

- Three viruses for the Apple machines emerged in 1981
  - Boot sector viruses
- Floppies of that time had the disk operating system (DOS) on them by default
  - Wrote it without malice

Brain

- ❑ First appeared in 1986
- ❑ More annoying than harmful
- ❑ A prototype for later viruses
- ❑ Not much reaction by users
- ❑ What it did
  1. Placed itself in boot sector (and other places)
  2. Screened disk calls to avoid detection
  3. Each disk read, checked boot sector to see if boot sector infected; if not, goto 1
- ❑ Brain did nothing malicious

Morris Worm

- First appeared in 1988
- What it tried to do
  - Determine where it could spread
  - Spread its infection
  - Remain undiscovered
- Morris claimed it was a test gone bad
- “Flaw“ in worm code — it tried to re-infect infected systems
  - Led to resource exhaustion
  - Adverse effect was like a so-called ***rabbit***

Morris Worm

- How to spread its infection?
- Tried to obtain access to machine by
  - User account password guessing
  - Exploited buffer overflow in fingerd
  - Exploited trapdoor in sendmail
- Flaws in fingerd and sendmail were well-known at the time, but not widely patched

Morris Worm

- Once access had been obtained to machine…
- “Bootstrap loader” sent to victim
  - Consisted of 99 lines of C code
- Victim machine compiled and executed code
- Bootstrap loader then fetched the rest of the worm
- Victim even authenticated the sender!

Morris Worm

- How to remain undetected?
- If transmission of the worm was interrupted, all code was deleted
- Code was encrypted when downloaded
- Downloaded code deleted after decrypting and compiling
- When running, the worm regularly changed its name and process identifier (PID)

Result of Morris Worm

- Shocked the Internet community of 1988
  - Internet of 1988 much different than today
- Internet designed to withstand nuclear war
  - Yet it was brought down by a graduate student!
  - At the time, Morris' father worked at NSA…
- Could have been much worse — not malicious
- Users who did not panic recovered quickest
- CERT began, increased security awareness
  - Though limited actions to improve security

Destructive Virus: Chernobyl (1998)

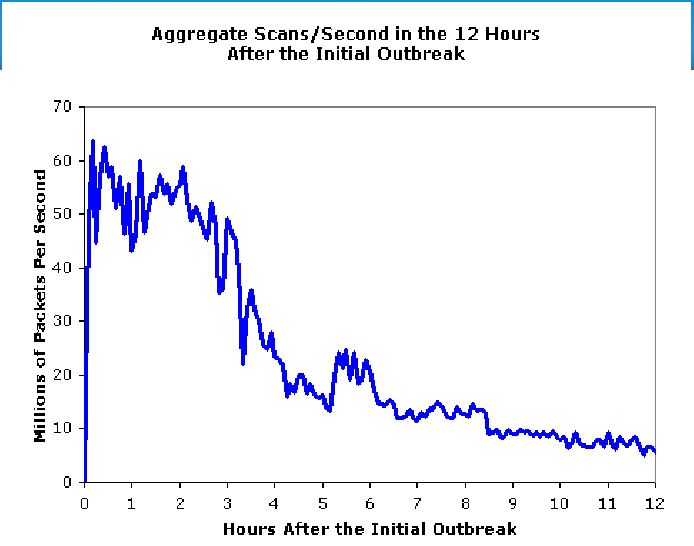
- Designed to inflict harm
  - Flash BIOS: would cause permanent hardware damage to vulnerable motherboards
  - Also overwrote first 2K sectors of each disk
    - Typically resulted in a loss of data and made it unbootable
- Previously believed that being benign was necessary for virus longevity
  - Chernobyl provided evidence to the contrary

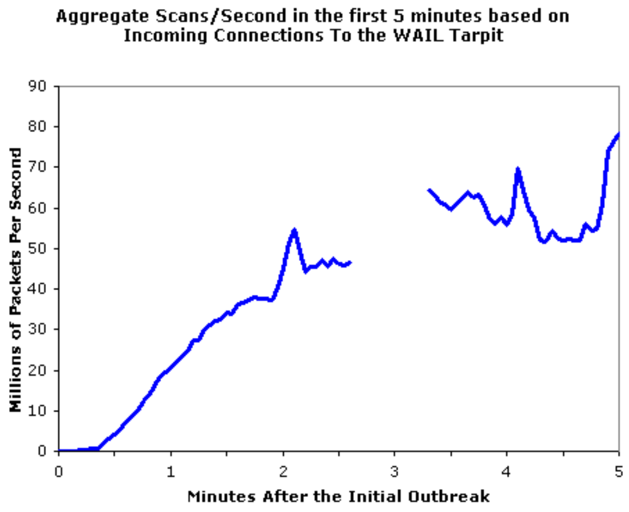
- Appeared in July 2001
- Infected more than **250,000 systems in about 15 hours**
- In total, infected 750,000 out of about 6,000,000 susceptible systems
- Exploited buffer overflow in Microsoft IIS server software
- Then monitored traffic on port 80 for other susceptible servers

- What it did
  - Day 1 to 19 of month: tried to spread infection
  - Day 20 to 27: distributed denial of service attack on `www.whitehouse.gov`
- Later versions (several variants)
  - Included trapdoor for remote access
  - Rebooted to flush worm, leaving only trapdoor
- Has been claimed that Code Red may have been "beta test for information warfare"

SQL Slammer

- Infected **250,000 systems in 10 minutes!**
- Code Red took 15 hours to do what Slammer did in 10 minutes
- At its peak, Slammer infections doubled every 8.5 seconds
- Slammer spread too fast
- "Burned out" available bandwidth





Outlines

- Mobile malcode Overview
- Viruses
- Worms

Viruses

- Definition from RFC 1135: A *virus* is a piece of code that inserts itself into a host, including operating systems, to propagate. It cannot run independently. It requires that its host program be run to activate it.
- On execution
  - Search for valid target files
    - Usually executable files
    - Often only infect uninfected files
  - Insert a copy into targeted files
    - When the target is executed, the virus starts running
- Only spread when contaminated files are moved from machine to machine
- Mature defenses available

Virus Operation

- virus phases:
  - dormant – waiting on trigger event
  - propagation – replicating to programs/disks
  - triggering – by event to execute payload
  - execution – of payload
- details usually machine/OS specific
  - exploiting features/weaknesses

Where do Viruses Live?

- Just about anywhere…
- Boot sector
  - Take control before anything else
- Memory resident
  - Stays in memory
- Applications, macros, data, etc.
- Library routines
- Compilers, debuggers, virus checker, etc.
  - These are particularly nasty!

Virus -- Macros

- Usually executable files: .com, .exe, .bat
- **Macro code** attached to some **data file**
- Interpreted by program using file
  - E.g., Word/Excel macros
  - Especially using auto command & command macros
- Code is now platform independent
- Is a major source of new viral infections
- Blur distinction between data and program files
- Classic trade-off: "ease of use" vs "security”
- Have improving security in Word etc
- Are no longer dominant virus threat

Variable Viruses

- Polymorphic viruses
  - Change with each infection
    - Executables virus code changing (macros: var name, line spacing, etc.)
    - Control flow permutations (rearrange code with goto's)
  - Attempt to defeat scanners
- Virus writing tool kits have been created to "simplify" creation of new viruses

Outlines

- Mobile malcode Overview
- Viruses
- Worms

Worms

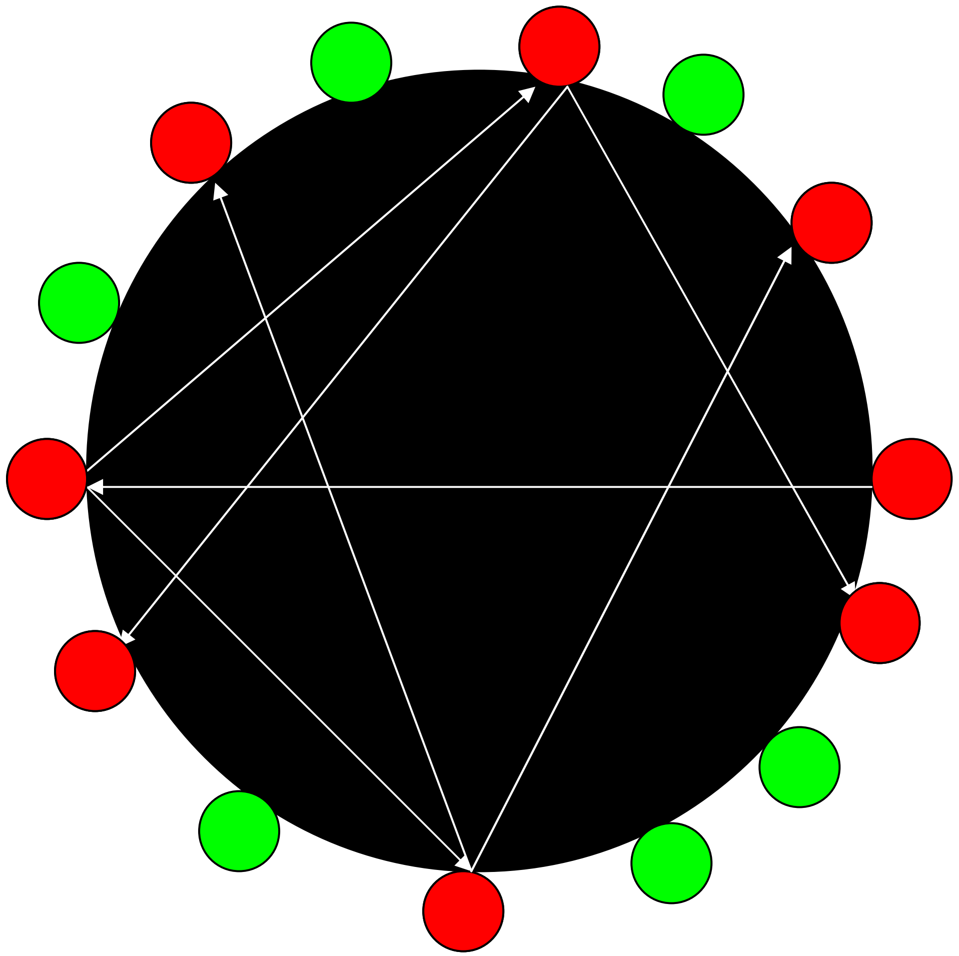
- Autonomous, active code that can replicate to remote hosts without any triggering
  - Replicating but not infecting program
- Because they propagate autonomously, they can **spread much more quickly** than viruses!
- Speed and general lack of user interaction make them the most significant threats
- using users distributed privileges or by exploiting system vulnerabilities
- subsequently used for further attacks

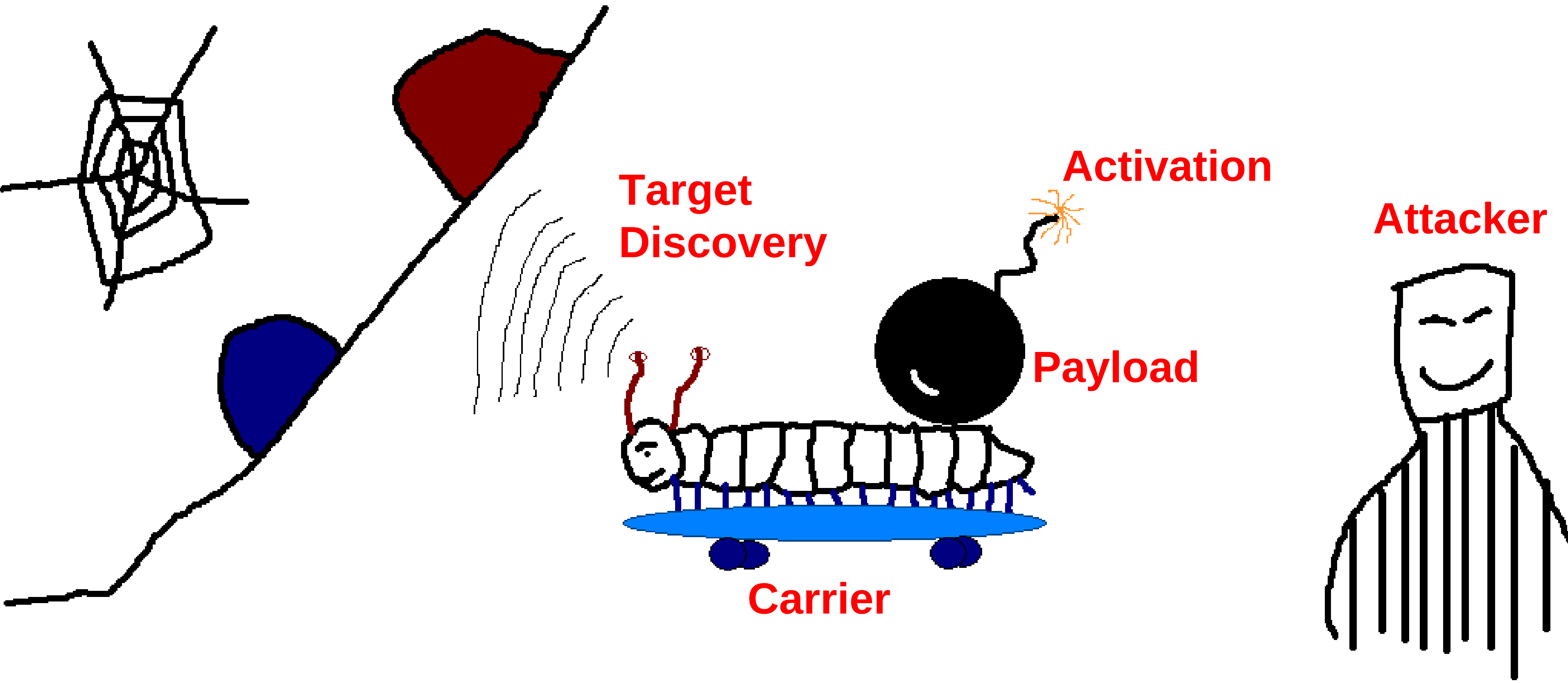
What is a worm?

Self propagating malcode.

Exponential speed.

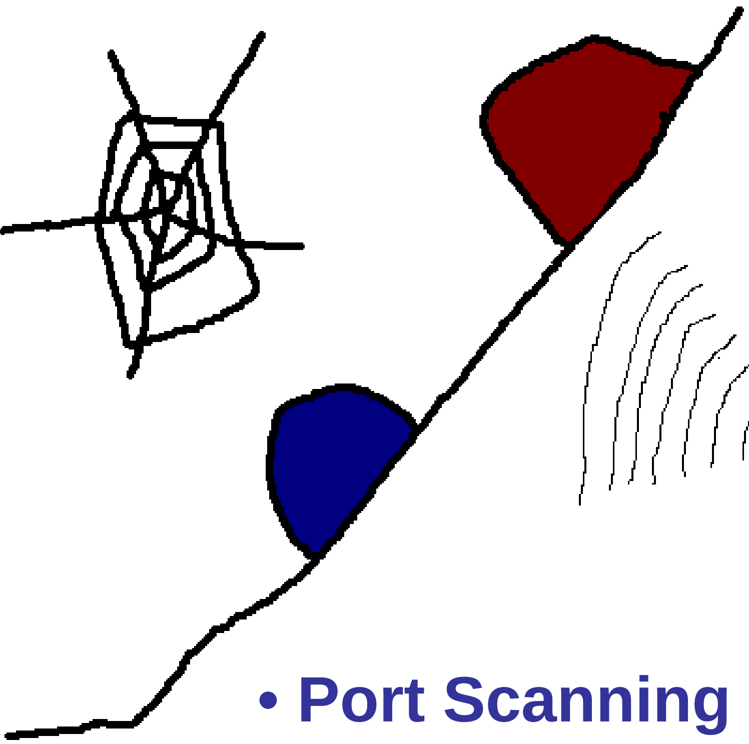
So far, Internet topology is not well exploited.

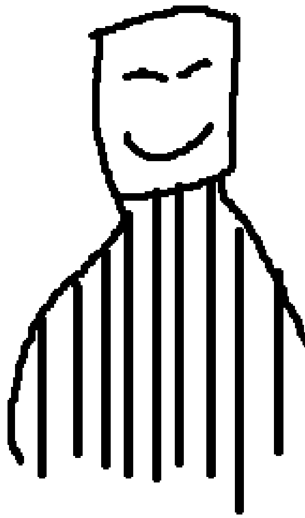




Worm Overview

Target Discovery

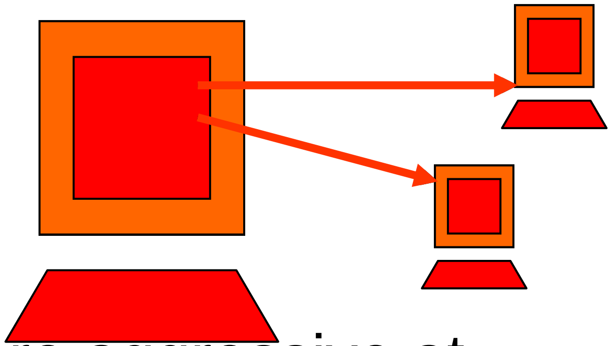




- **Port Scanning**
  - Sequential: working through an address block
  - Random
- **Target Lists**
  - Externally generated through Meta servers
  - Internal target list
  - Passive worms

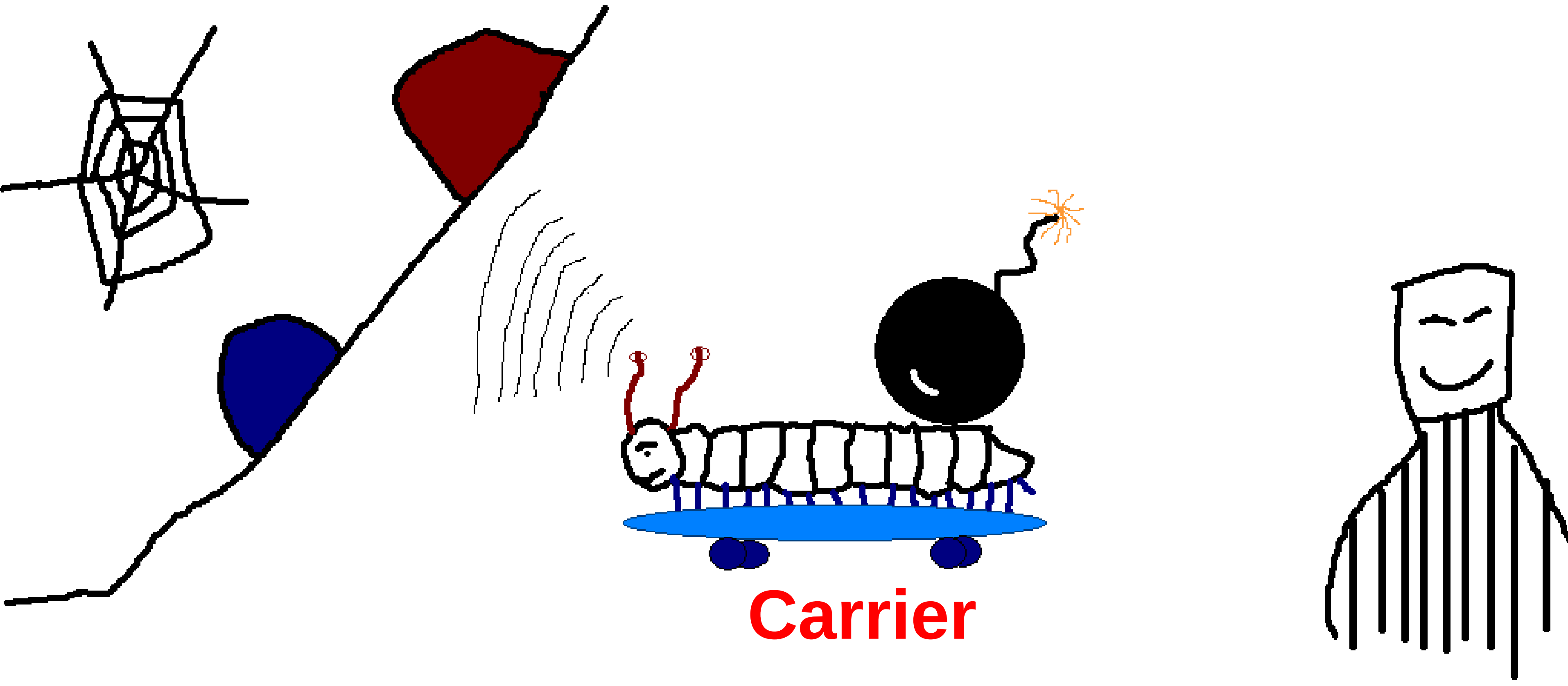
Internal Target Lists: Topological Information

- Look for local information to find new targets
  - URLs on disk and in caches
  - Mail addresses
  - .ssh/known_hosts
- Ubiquitous in mail worms
  - More recent mail worms are more aggressive at finding new addresses
- Basis of the Morris worm (1988)
  - Address space was too sparse for scanning to work



Passive Worms

- Wait for information about other targets
  E.g., CRclean, an anti-CodeRed II worm
  - Wait for Code Red, respond with counterattack
  - Remove Code Red II and install itself on the machine
- Speed is highly variable
  - Depends on normal communication traffic
- Very high stealth
  - Have to detect the act of infection, not target selection

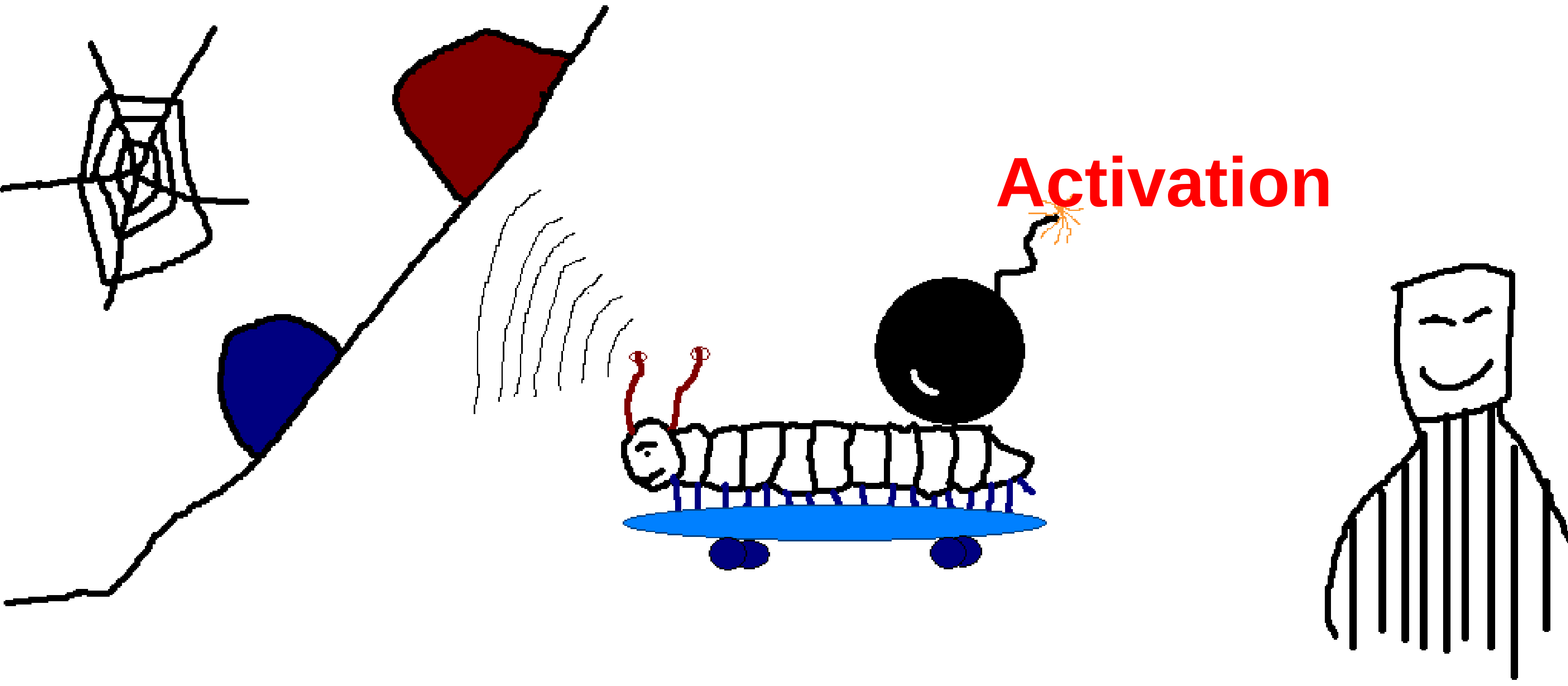


- **Self-Carried**

  Transmit itself as part of the infection process

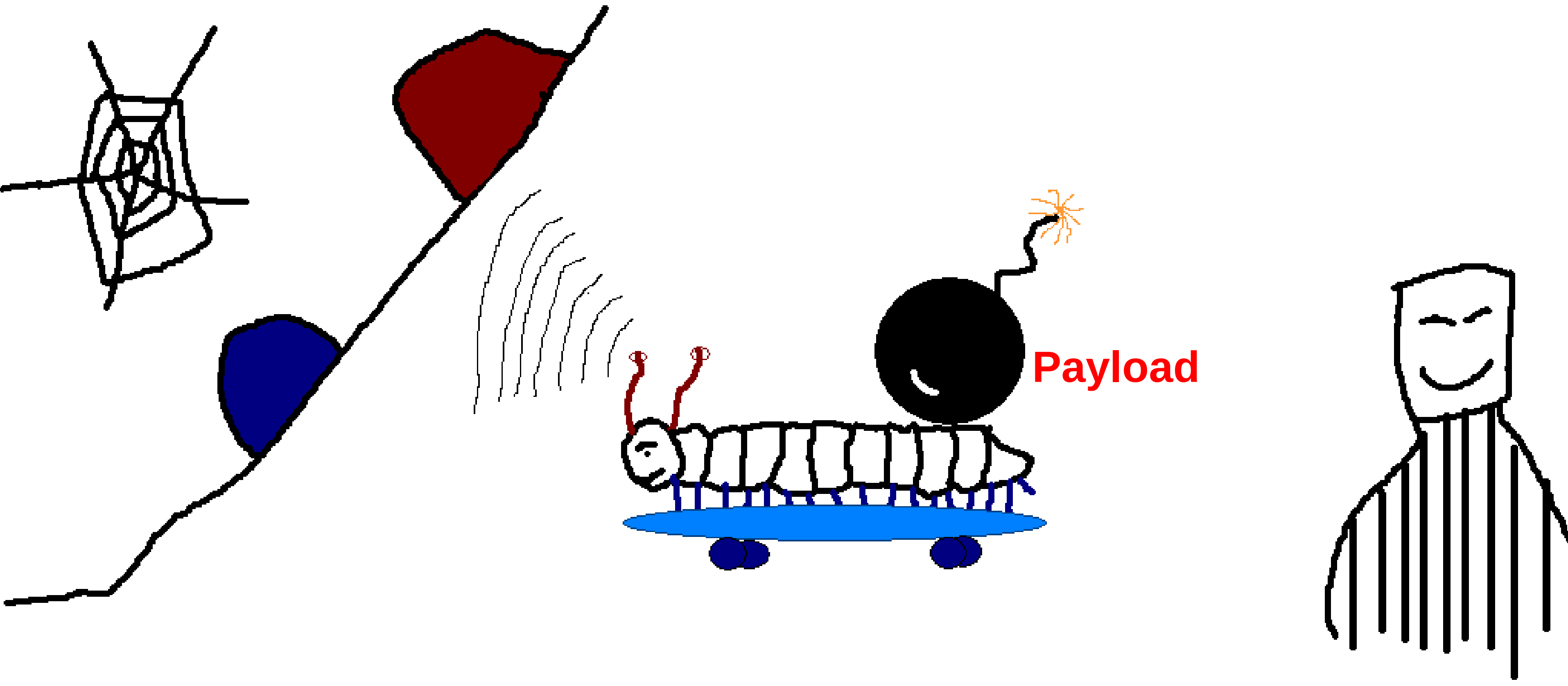
- **Second Channel**

  E.g. blaster worm use RPC to exploit, but use TFTP to download the whole virus body


Activation

Activation

- Human Activation
  - Needs social engineering, especially for email worms
    - Melissa – "Attached is an important message for you!"
    - Iloveyou – "Open this message to see who loves you!"
- Human activity-based activation
  - E.g. logging in, rebooting (Nimda's secondary propagation)
- Scheduled process activation
  - E.g. updates, backup etc.
- Self Activation
  - E.g. Code Red exploit the IIS web servers

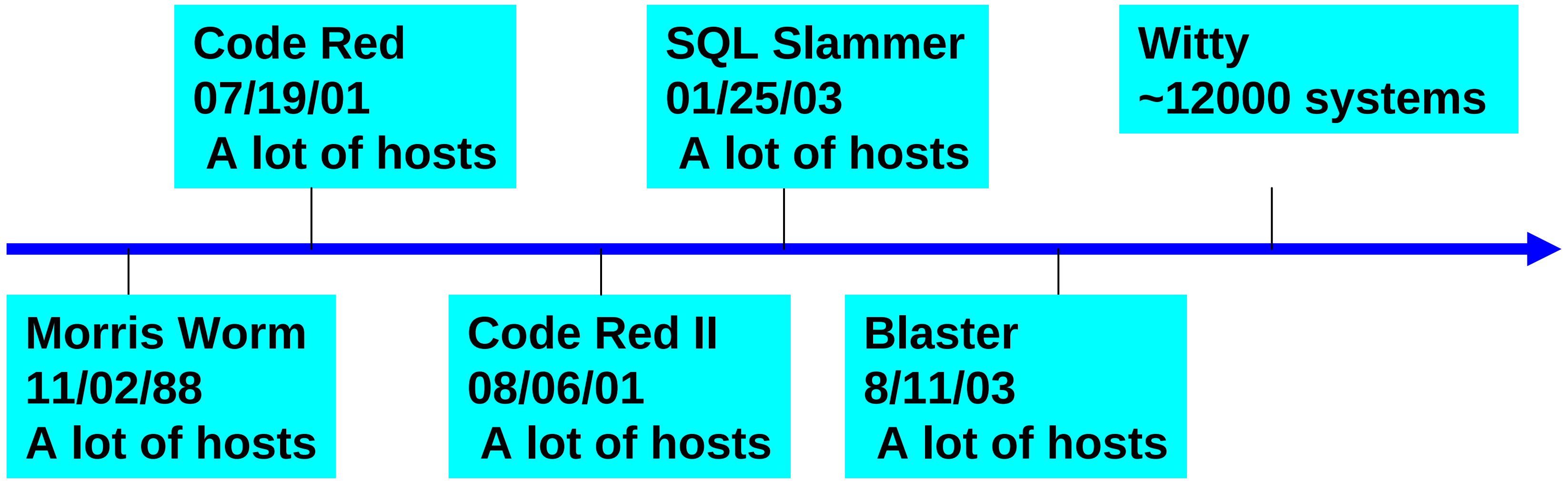

Payload

Payloads

- None/nonfunctional
  - Most common
  - Still can have significant effects through traffic and machine load (e.g., Morris worm, Slammer, …)
- Internet Remote Control
  - Code Red II open backdoor on victim machines: anyone with a web browser can execute arbitrary code
- Internet Denial of Service (DOS)
  - E.g., Code Red, Yaha
- Data Collection
- Data Damage: Chernobyl , Klez
- Worm maintenance

- Experimental Curiosity, e.g., I Love You worm
- Pride and Power
- Commercial Advantage
- Extortion and Criminal Gain
- Terrorism
- Cyber Warfare

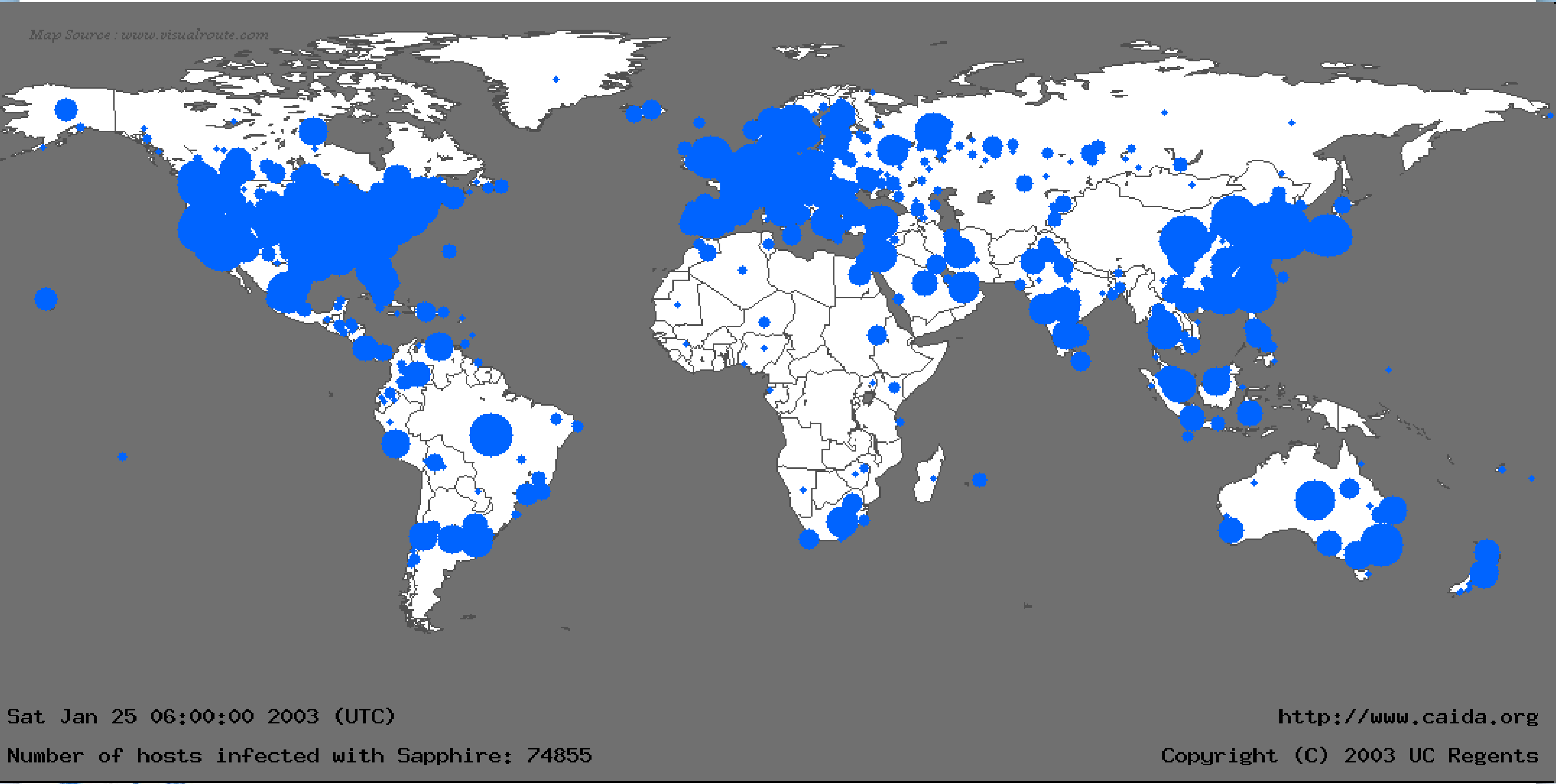
A little history



Benign effects:
- SQL Slammer congests the network
- Morris worm crashes hosts
- … …

Destructive effects:
- Code Red defaces web pages
- Witty overwrites a random disk block
- … …

The Spread of the SQL Slammer Worm



Sat Jan 25 06:00:00 2003 (UTC)

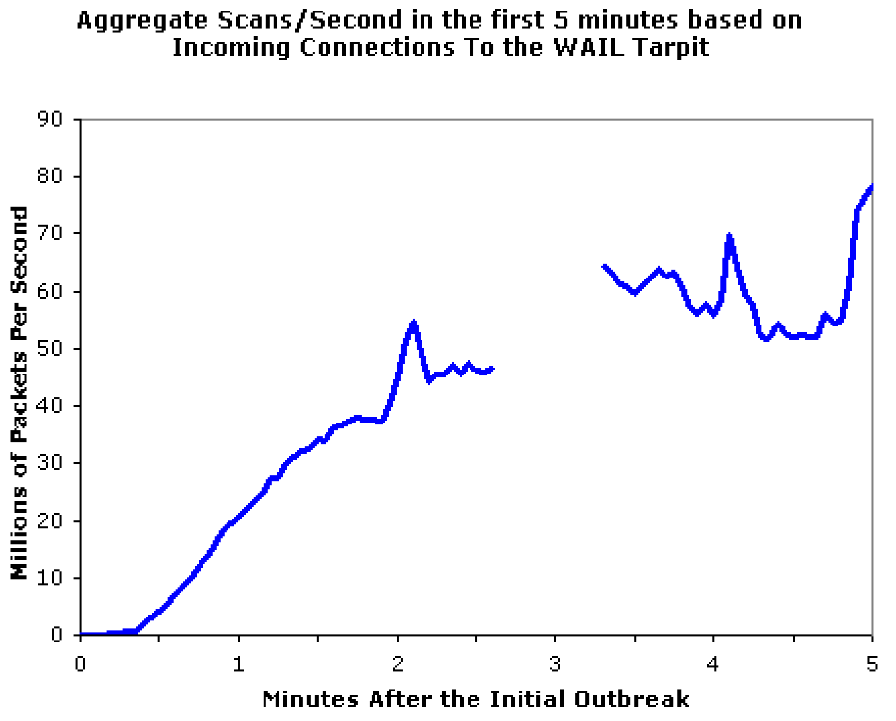
Number of hosts infected with Sapphire: 74855

http://www.caida.org

Copyright (C) 2003 UC Regents

How Fast was Slammer?

- Infected ~75,000 machines in 10 minutes
- Full scanning rate in ~3 minutes
  - >55 Million IPs/s
- Initial doubling rate was about every 8.5 seconds
  - Local saturations occur in <1 minute



Malware Detection

- Three common methods
  - Signature detection - Look for patterns
  - Change detection - Integrity Checking
  - Anomaly detection - Look for bad behavior
- We'll briefly discuss each of these
  - And consider advantages and disadvantages of each

Signature Detection

- A signature is a string of bits found in software (or could be a hash value)
- Suppose that a virus has signature 0x23956a58bd910345
- We can search for this signature in all files
- If we find the signature are we sure we've found the virus?
  - No, same signature could appear in other files
  - But at random, chance is very small: $1/2^{64}$
  - Software is not random, so probability is higher

Signature Detection

- Advantages
  - Effective on “traditional” malware
  - Minimal burden for users/administrators
- Disadvantages
  - Signature file can be large (10,000’s)…
  - …making scanning slow
  - Signature files must be kept up to date
  - Cannot detect unknown viruses
  - Cannot detect some new types of malware
- By far the most popular detection method

Change Detection

- Viruses must live somewhere on system
- If we detect that a file has changed, it may be infected
- How to detect changes?
  - Hash files and (securely) store hash values
  - Recompute hashes and compare
  - If hash value changes, file **might** be infected

Change Detection

- Advantages
  - Virtually no false negatives
  - Can even detect previously unknown malware
- Disadvantages
  - Many files change — and often
  - Many false alarms (false positives)
  - Heavy burden on users/administrators
  - If suspicious change detected, then what?
  - Might still need signature-based system

Anomaly Detection

- Monitor system for anything “unusual” or “virus-like” or potentially malicious
- What is unusual?
  - Files change in some unusual way
  - System misbehaves in some way
  - Unusual network activity
  - Unusual file access, etc., etc., etc.
- But must first define “normal”
  - And normal can change!

Anomaly Detection

- Advantages
  - Chance of detecting unknown malware
- Disadvantages
  - Unproven in practice
  - Trudy can make abnormal look normal (go slow)
  - Must be combined with another method (such as signature detection)
- Also popular in intrusion detection (IDS)
- A difficult unsolved (unsolvable?) problem
  - As difficult as AI?

Future of Malware

- Polymorphic and metamorphic malware
- Fast replication/Warhol worms
- Flash worms, Slow worms, etc.
- Future is bright for malware
  - Good news for the bad guys…
  - …bad news for the good guys
- Future of malware detection?