

Part IV: Software

Why Software?

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

Chapter 11:

Software Flaws and Malware

If automobiles had followed the same development cycle as the computer,
a Rolls-Royce would today cost \$100, get a million miles per gallon,
and explode once a year, killing everyone inside.

—Robert X. Cringely

My software never has bugs. It just develops random features.

—Anonymous

Bad Software is Ubiquitous

- ❑ NASA Mars Lander (cost \$165 million)
 - Crashed into Mars due to...
 - ...error in converting English and metric units of measure
 - Believe it or not
- ❑ Denver airport
 - Baggage handling system —very buggy software
 - Delayed airport opening by 11 months
 - Cost of delay exceeded \$1 million/day
 - What happened to person responsible for this fiasco?
- ❑ MV-22 Osprey
 - Advanced military aircraft
 - Faulty software can be fatal

Software Issues

Alice and Bob

- ❑ Find bugs and flaws by accident
- ❑ Hate bad software...
- ❑ ...but they learn to live with it
- ❑ Must make bad software work

Trudy

- ❑ Actively looks for bugs and flaws
- ❑ Likes bad software...
- ❑ ...and tries to make it misbehave
- ❑ Attacks systems via bad software

Complexity

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

System	Lines of Code (LOC)
Netscape	17 million
Space Shuttle	10 million
Linux kernel 2.6.0	5 million
Windows XP	40 million
Mac OS X 10.4	86 million
Boeing 777	7 million

- 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/10,000 LOC
- ❑ **Do the math**
 - Typical computer: 3k exe's of 100k LOC each
 - Conservative estimate: 50 bugs/exe
 - Implies about 150k bugs per computer
 - So, 30,000-node network has 4.5 billion bugs
 - Maybe only 10% of bugs security-critical and only 10% of those remotely exploitable
 - Then "only" 45 million critical security flaws!

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



Example

```
char array[10];  
for(i = 0; i < 10; ++i)  
    array[i] = `A`;   
array[10] = `B`;
```

- ❑ This program has an **error**
- ❑ This error might cause a **fault**
 - Incorrect internal state
- ❑ If a fault occurs, it might lead to a **failure**
 - Program behaves incorrectly (external)
- ❑ We use the term **flaw** for all of the above

Secure Software

- ❑ In software engineering, try to ensure that a program does what is intended
- ❑ *Secure* software engineering requires that software **does what is intended...**
- ❑ **...and nothing more**
- ❑ Absolutely secure software? Dream on...
 - Absolute security *anywhere* is impossible
- ❑ How can we manage software risks?

Program Flaws

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

Buffer Overflow



Attack Scenario

- ❑ Users enter data into a Web form
- ❑ Web form is sent to server
- ❑ Server writes data to array called buffer, without checking length of input data
- ❑ Data “overflows” buffer
 - Such overflow might enable an attack
 - If so, attack could be carried out by anyone with Internet access

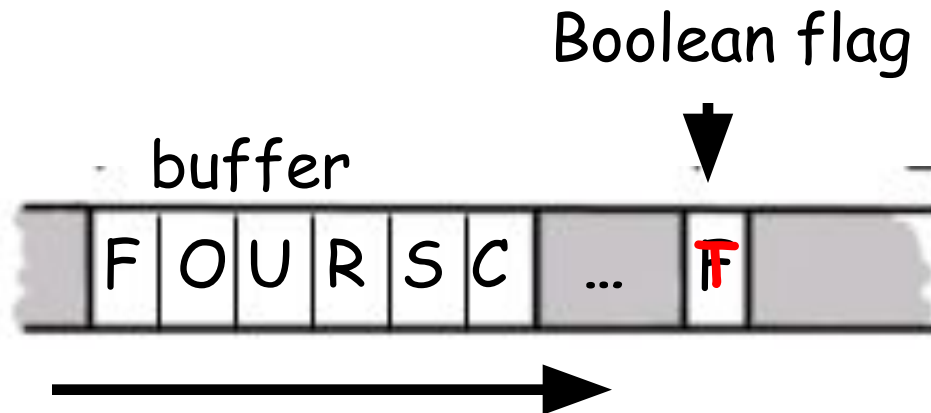
Buffer Overflow

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

- ❑ **Q:** What happens when code 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
 - Or program could work just fine

Simple Buffer Overflow

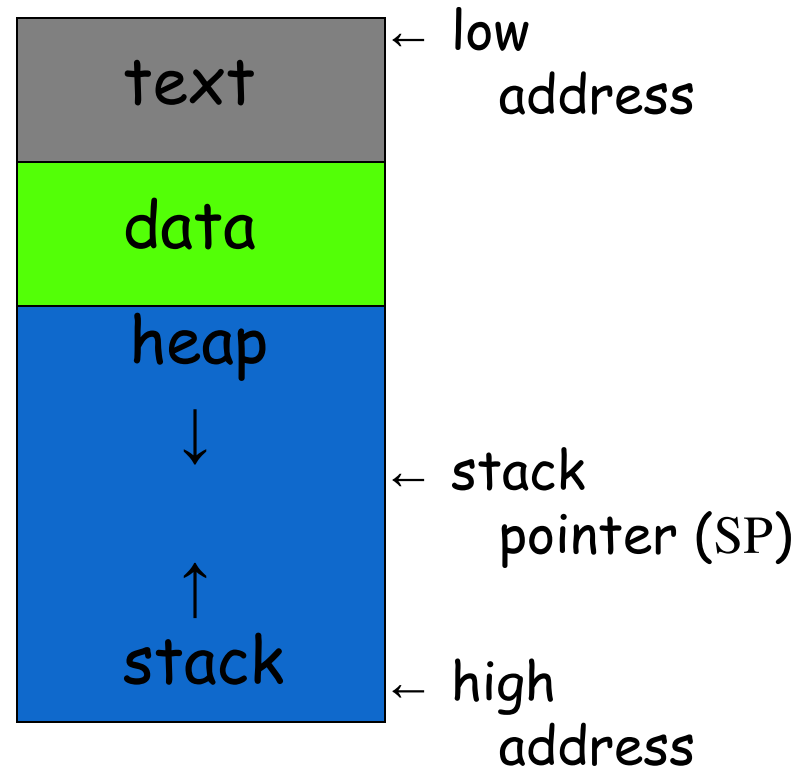
- ❑ Consider boolean flag for authentication
- ❑ Buffer overflow could overwrite flag allowing anyone to authenticate



- ❑ In some cases, Trudy need not be so lucky as in this example

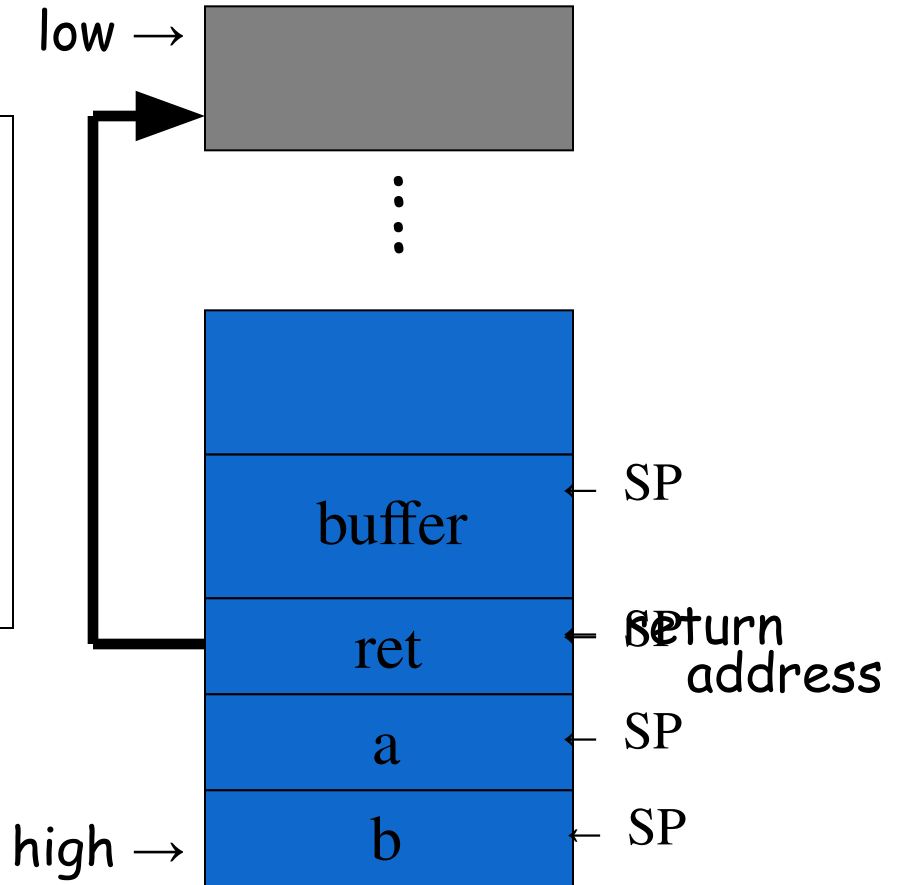
Memory Organization

- ❑ **Text** —code
- ❑ **Data** —static variables
- ❑ **Heap** —dynamic data
- ❑ **Stack** —“scratch paper”
 - Dynamic local variables
 - Parameters to functions
 - Return address



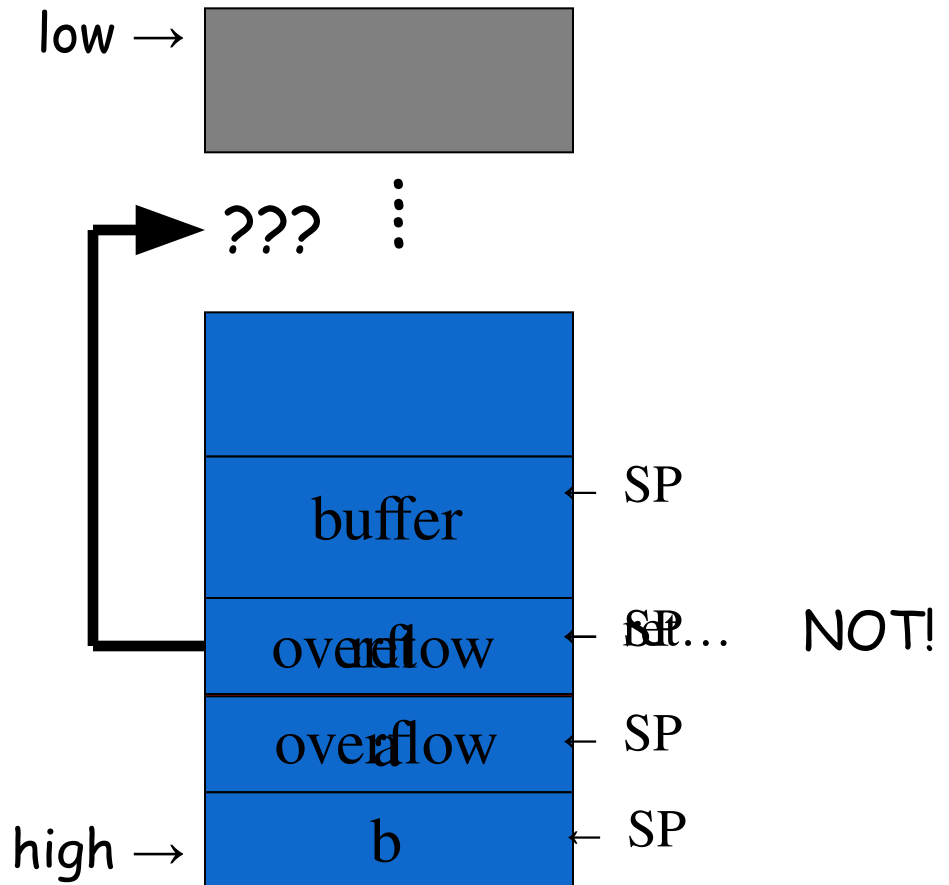
Simplified Stack Example

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



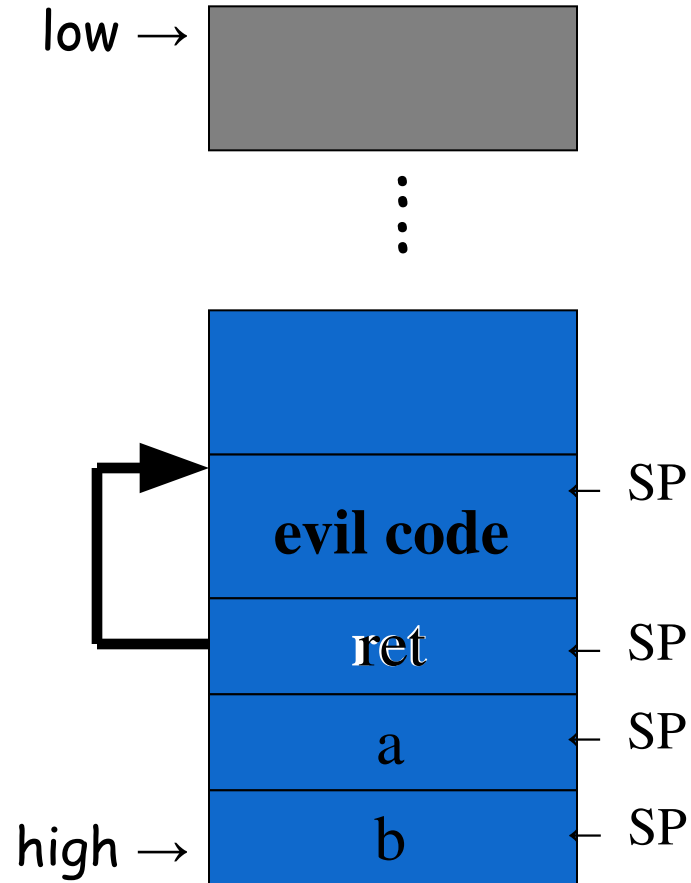
Smashing the Stack

- ❑ 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...
 - ...on your machine

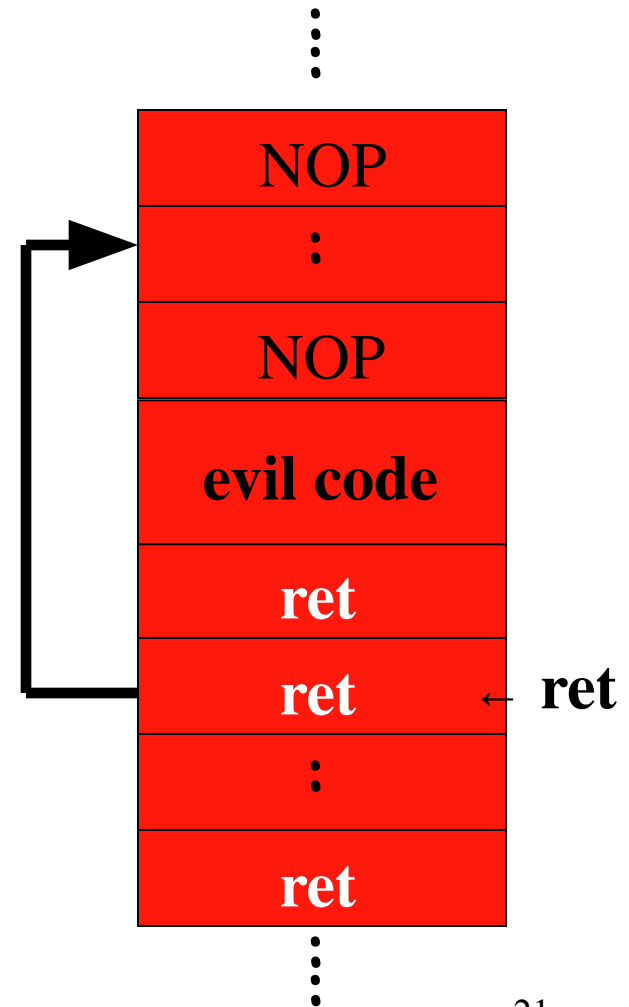


Smashing the Stack

- ❑ Trudy may not know...
 - 1) Address of evil code
 - 2) Location of **ret** on stack

- ❑ Solutions

- 1) Precede evil code with NOP "landing pad"
- 2) Insert ret many times

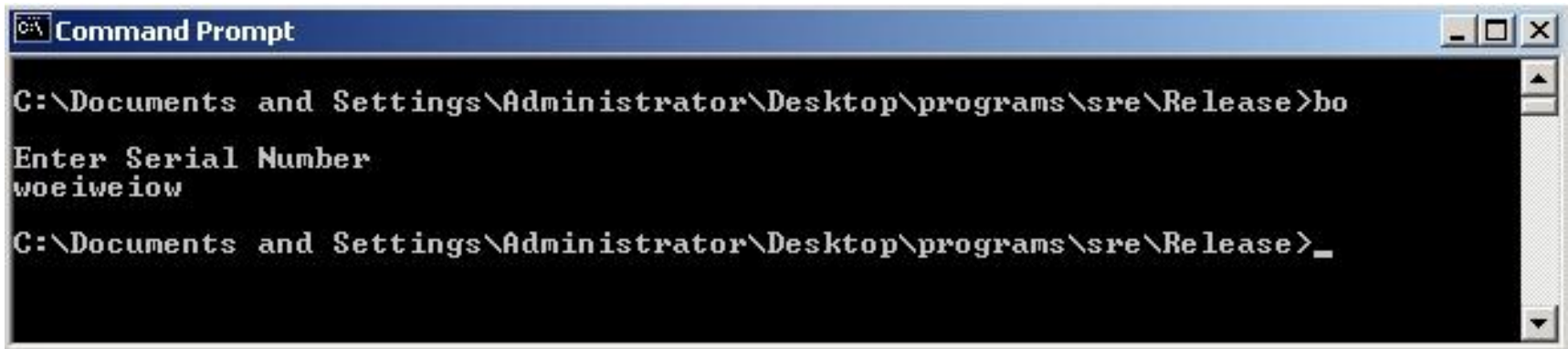


Stack Smashing Summary

- ❑ A buffer overflow must exist in the code
- ❑ Not all buffer overflows are exploitable
 - Things must align properly
- ❑ If exploitable, attacker can **inject code**
- ❑ Trial and error is likely required
 - Fear not, lots of help is available online
 - [Smashing the Stack for Fun and Profit](#), Aleph One
- ❑ Stack smashing is “attack of the decade”...
 - ...for many recent decades
 - Also heap & integer overflows, format strings, etc.

Stack Smashing Example

- ❑ Suppose program asks for a serial number that Trudy does not know
- ❑ Also, Trudy does **not** have source code
- ❑ Trudy only has the executable (exe)



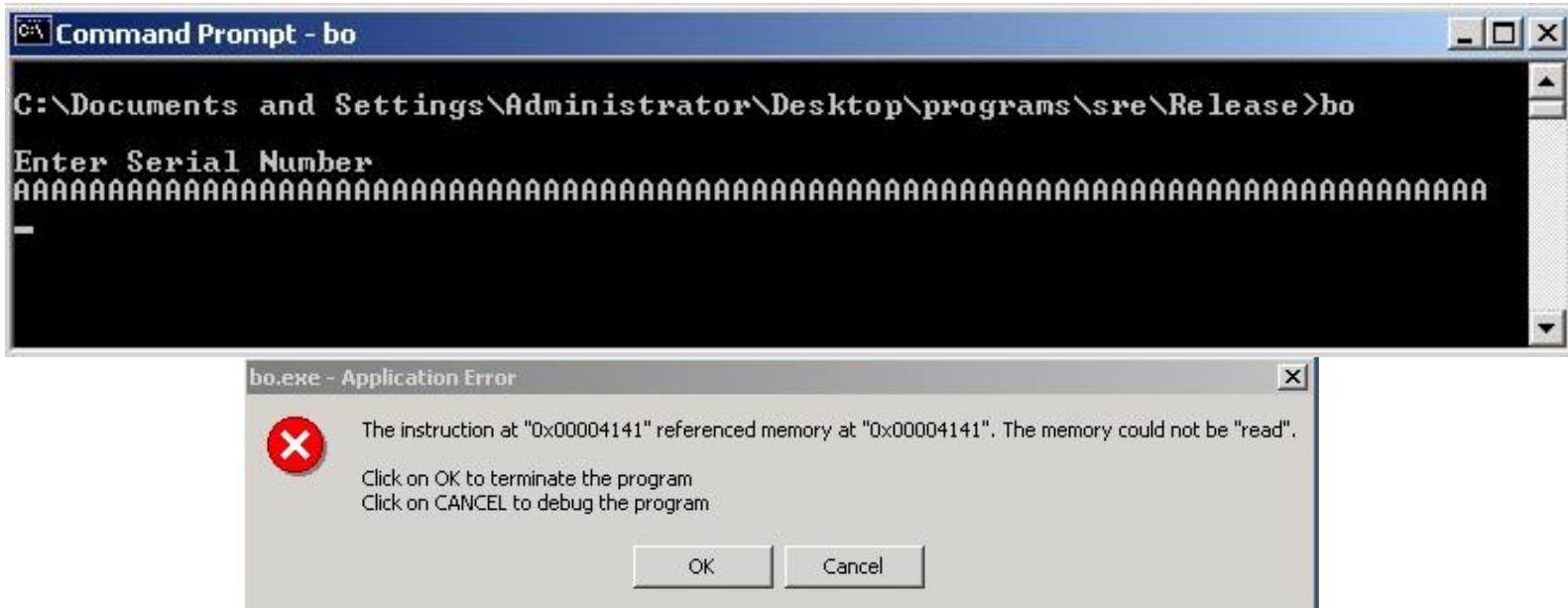
The screenshot shows a Windows Command Prompt window titled "Command Prompt". The current directory is "C:\Documents and Settings\Administrator\Desktop\programs\sre\Release". The user has entered the command "bo", which has triggered the program to prompt for a "Serial Number". The user has entered "woeiweiw", which is an incorrect serial number. The prompt is now waiting for the next input.

```
C:\Documents and Settings\Administrator\Desktop\programs\sre\Release>bo
Enter Serial Number
woeiweiw
C:\Documents and Settings\Administrator\Desktop\programs\sre\Release>_
```

- ❑ Program quits on incorrect serial number

Buffer Overflow Present?

- ❑ By trial and error, Trudy discovers apparent buffer overflow



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

Disassemble Code

- Next, disassemble bo.exe to find

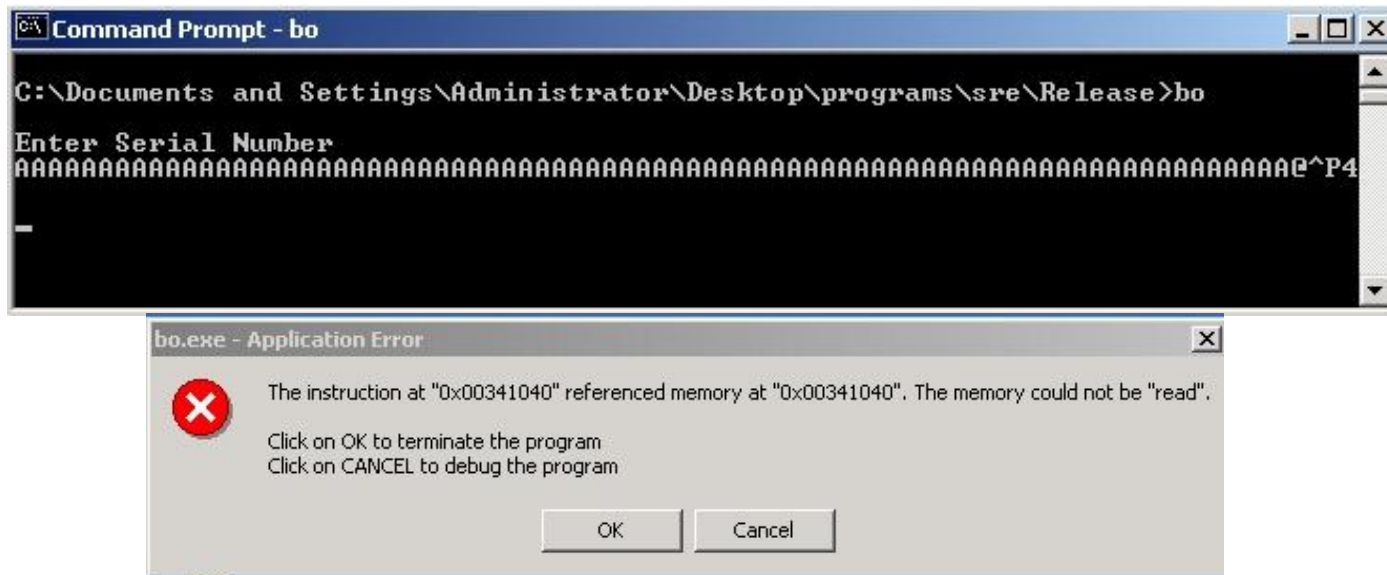
```
.text:00401000
.text:00401000
.text:00401003
.text:00401008
.text:0040100D
.text:00401011
.text:00401012
.text:00401017
.text:0040101C
.text:0040101E
.text:00401022
.text:00401027
.text:00401028
.text:0040102D
.text:00401030
.text:00401032
.text:00401034
.text:00401039
.text:0040103E

sub     esp, 1Ch
push    offset aEnterSerialNum ; "\nEnter Serial Number\n"
call    sub_40109F
lea     eax, [esp+20h+var_1C]
push    eax
push    offset aS                ; "%S"
call    sub_401088
push    8
lea     ecx, [esp+2Ch+var_1C]
push    offset aS123n456 ; "S123N456"
push    ecx
call    sub_401050
add     esp, 18h
test    eax, eax
jnz     short loc_401041
push    offset aSerialNumberIs ; "Serial number is correct.\n"
call    sub_40109F
add     esp, 4
```

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

Buffer Overflow Attack


- Find that, in *ASCII*, 0x401034 is "@^P4"



- ❑ Byte order is reversed? What the ... ?
- ❑ X86 processors are "little-endian"

Overflow Attack, Take 2

- ❑ Reverse the byte order to "4^P@" and...



```
C:\Documents and Settings\Administrator\Desktop\programs\sre\Release>bo
Enter Serial Number
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA4^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
- ❑ What just happened?
 - Overwrote return address on the stack

Buffer Overflow

- ❑ Trudy did **not** require access to the source code
- ❑ Only tool used was a disassembler to determine address to jump to
- ❑ Find desired address by trial and error?
 - Necessary if attacker does not have exe
 - For example, a remote attack

Source Code

- ❑ Source code for buffer overflow example
- ❑ Flaw easily exploited by attacker...
- ❑ ...without access to source code!

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

main()
{
    char in[75];

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

    scanf("%s", in);

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

Stack Smashing Defenses

- ❑ 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, for example)
- ❑ Use a **canary**
- ❑ Address space layout randomization (**ASLR**)
- ❑ Use **safe languages** (Java, C#)
- ❑ Use **safer C functions**
 - For unsafe functions, safer versions exist
 - For example, `strncpy` instead of `strcpy`

Stack Smashing Defenses

□ Canary

- Run-time stack check
- Push canary onto stack
- Canary value:
 - Constant `0x000aff0d`
 - Or, may depends on `ret`



Microsoft's Canary

- ❑ Microsoft added **buffer security check** feature to C++ with /GS compiler flag

- Based on canary (or "security cookie")

Q: What to do when canary dies?

A: Check for user-supplied "handler"

- ❑ Handler shown to be subject to attack
 - Claimed that attacker can specify handler code
 - If so, formerly "safe" buffer overflows become exploitable when /GS is used!

ASLR

- ❑ Address Space Layout Randomization
 - Randomize place where code loaded in memory
- ❑ Makes most buffer overflow attacks probabilistic
- ❑ Windows Vista used 256 random layouts
 - So about 1/256 chance buffer overflow works
- ❑ Similar thing in Mac OS X and other OSs
- ❑ Attacks against Microsoft's ASLR do exist
 - Possible to "de-randomize"

Buffer Overflow

- ❑ A major security threat yesterday, today, and tomorrow
- ❑ The good news?
 - It is possible to reduce overflow attacks (safe languages, NX bit, ASLR, education, etc.)
- ❑ The bad news?
 - Buffer overflows will exist for a long time
 - Why? Legacy code, bad development practices, clever attacks, etc.

Incomplete Mediation



Input Validation

- ❑ Consider: `strcpy(buffer, argv[1])`
- ❑ A buffer overflow occurs if
 $\text{len}(\text{buffer}) < \text{len}(\text{argv}[1])$
- ❑ 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=10&shipping=5&total=25`

Incomplete Mediation

- ❑ Linux kernel
 - Research revealed many buffer overflows
 - Lots of these 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 for attackers too!

Race Conditions

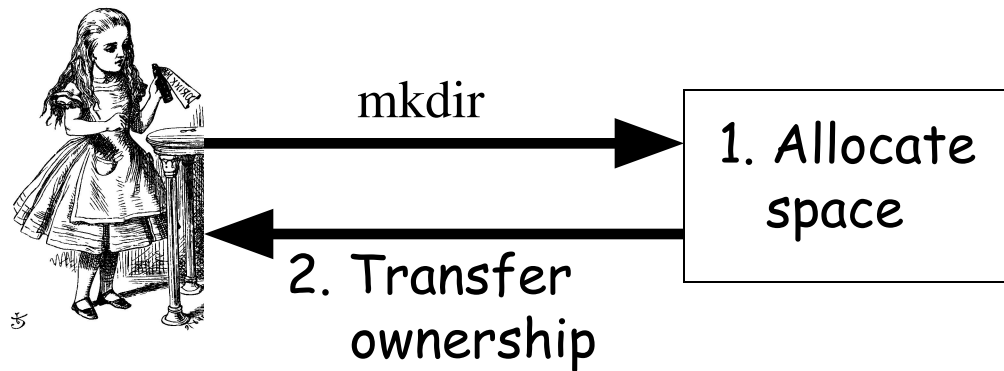


Race Condition

- ❑ 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
- ❑ Example: Unix `mkdir`

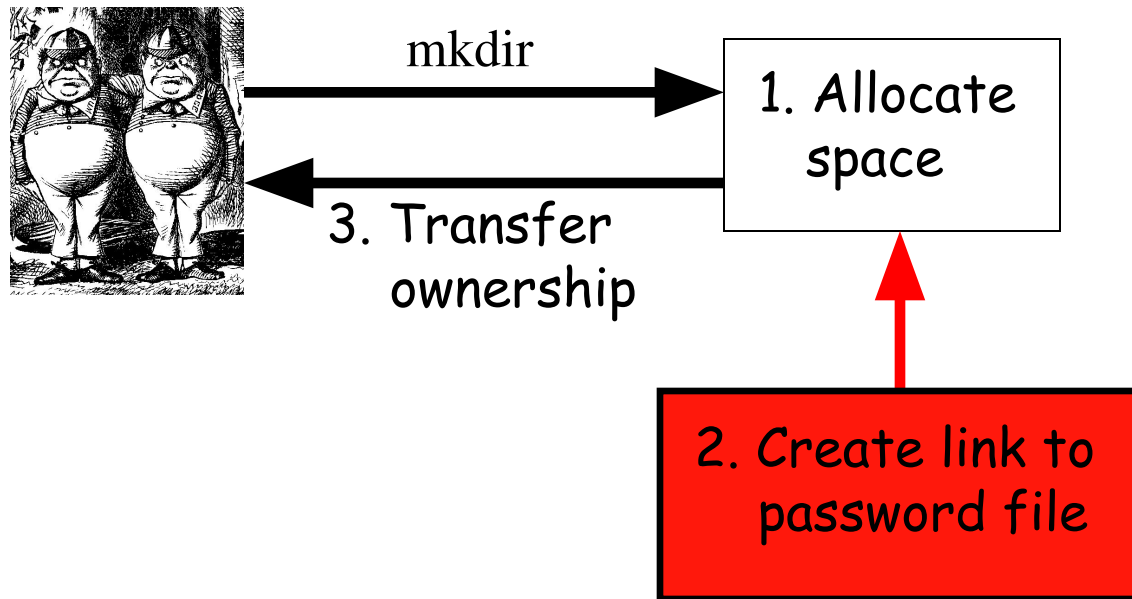
mkdir Race Condition

- ❑ mkdir creates new directory
- ❑ How mkdir is supposed to work



mkdir Attack

❑ The mkdir **race condition**



- ❑ Not really a "race"
 - But attacker's timing is critical

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

Malware

Malicious Software

- ❑ Malware is not new...
 - Fred Cohen's initial virus work in 1980's
- ❑ Types of malware (no standard definition)
 - **Virus** —passive propagation
 - **Worm** —active propagation
 - Trojan horse —unexpected functionality
 - Trapdoor/backdoor —unauthorized access
 - Rabbit —exhaust system resources
 - Spyware —steals info, such as passwords
 - Botnets —malware for hire
 - Ransomware —Malware encrypts user data

Where do Viruses Live?

- ❑ They live just about anywhere, such as...
- ❑ Boot sector
 - Take control before anything else
- ❑ Memory resident
 - Stays in memory
- ❑ Applications, macros, data, etc.
- ❑ Library routines
- ❑ Compilers, debuggers, virus checker, etc.
 - These would be particularly nasty!

Malware Examples

- ❑ Brain virus (1986)
- ❑ Morris worm (1988)
- ❑ Code Red (2001)
- ❑ SQL Slammer (2004)
- ❑ Stuxnet (2010)
- ❑ Botnets, Ransomware (current)
- ❑ Future of malware?

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 really malicious

Morris Worm

- ❑ Appeared in 1988
- ❑ What it tried to do
 - Determine where it could spread, then...
 - ...spread its infection and...
 - ...remain undiscovered
- ❑ Morris claimed his worm had a bug!
 - It tried to re-infect infected systems
 - Led to resource exhaustion
 - Effect was like a so-called rabbit

How Morris Worm Spread

- ❑ Obtained access to machines by...
 - User account password guessing
 - Exploit **buffer overflow** in fingerd
 - Exploit **trapdoor** in sendmail
- ❑ Flaws in fingerd and sendmail were well-known, but not widely patched

Bootstrap Loader

- ❑ Once Morris worm got access...
- ❑ "Bootstrap loader" sent to victim
 - 99 lines of C code
- ❑ Victim compiled and executed code
- ❑ Bootstrap loader fetched the worm
- ❑ Victim **authenticated** sender
 - Don't want user to get a bad worm...

How to Remain Undetected?

- ❑ If transmission interrupted, all code deleted
- ❑ Code encrypted when downloaded
- ❑ Code deleted after decrypt/compile
- ❑ When running, worm regularly changed name and process identifier (PID)

Morris Worm: Bottom Line

- ❑ Shocked the Internet community of 1988
 - Internet of 1988 *much* different than today
- ❑ Internet designed to survive nuclear war
 - Yet, brought down by one graduate student!
 - At the time, Morris' father worked at NSA...
- ❑ Could have been much worse
- ❑ Result? CERT, more security awareness
- ❑ But should have been a wakeup call

Code Red Worm

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

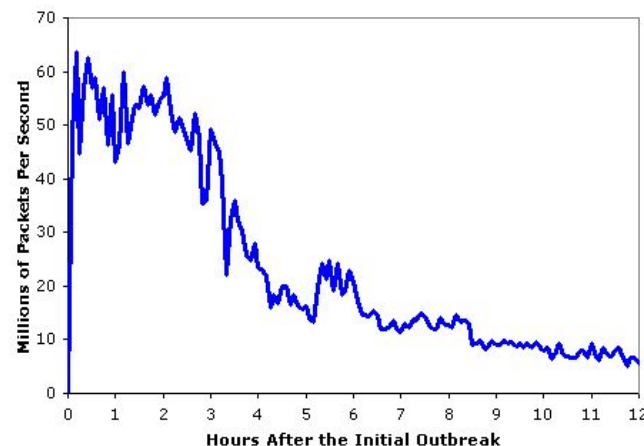
Code Red: What it Did

- ❑ Day 1 to 19 of month: spread its infection
- ❑ Day 20 to 27: distributed denial of service attack (DDoS) on `www.whitehouse.gov`
- ❑ Later version (several variants)
 - Included trapdoor for remote access
 - Rebooted to flush worm, leaving only trapdoor
- ❑ Some said it was "beta test for info warfare"
 - But, no evidence to support this

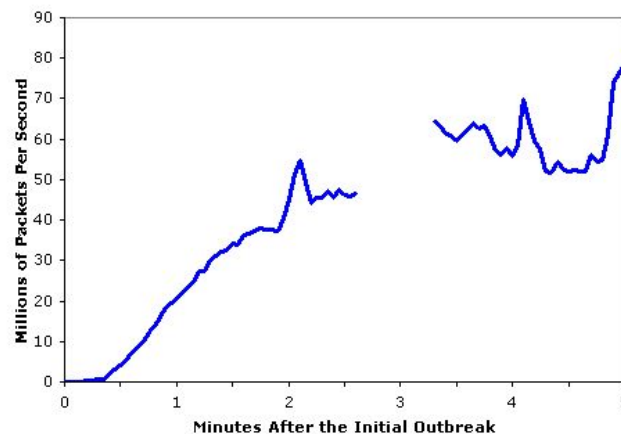
SQL Slammer

- Infected **75,000 systems in 10 minutes!**
- At its peak, infections doubled every 8.5 seconds
- Spread "too fast"...
- ...so it "burned out" available bandwidth

Aggregate Scans/Second in the 12 Hours After the Initial Outbreak



Aggregate Scans/Second in the first 5 minutes based on Incoming Connections To the WAIL Tarpit



Why was Slammer Successful?

- ❑ Worm size: **one 376-byte UDP packet**
- ❑ Firewalls often let one packet thru
 - Then monitor ongoing "connections"
- ❑ Expectation was that much more data required for an attack
 - So no need to worry about 1 small packet
- ❑ Slammer defied "experts"

Stuxnet

- ❑ Malware for information warfare...
- ❑ Discovered in 2010
 - Origins go back to 2008, or earlier
- ❑ Apparently, targeted Iranian nuclear processing facility
 - Reprogrammed specific type of PLC
 - Changed speed of centrifuges, causing damage to 1000 (or more) of them

Stuxnet

- ❑ Many advanced features including...
 - Infect system via removable drives — able to get behind “airgap” firewalls
 - Used 4 unpatched MS vulnerabilities
 - Updates via P2P over LAN
 - Contact C&C server for code/updates
 - Includes a Windows rootkit for stealth
 - Significant exfiltration/recon capability
 - Used a compromised private key


Malware Related to Stuxnet

- ❑ Duqu (2011)
 - Likely that developers had access to Stuxnet source code
 - Apparently, used mostly for info stealing
- ❑ Flame (2012)
 - May be “most complex” malware ever
 - Sophisticated spyware mechanisms

Malware Detection

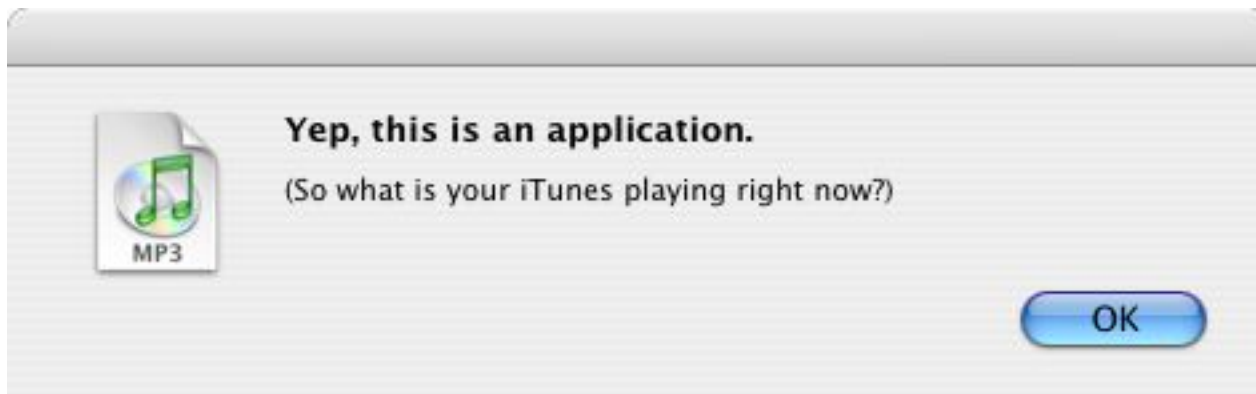
- ❑ Three common detection methods
 - Signature detection
 - Change detection
 - Anomaly detection
- ❑ We briefly discuss each of these
 - And consider advantages...
 - ...and disadvantages

Trojan Horse Example

- ❑ Trojan: unexpected functionality
- ❑ Prototype trojan for the Mac
- ❑ File icon for freeMusic.mp3:
freeMusic.mp3
- ❑ For a real mp3, double click on icon
 - iTunes opens
 - Music in mp3 file plays
- ❑ But for freeMusic.mp3, unexpected results...

Mac Trojan

- ❑ Double click on freeMusic.mp3
 - iTunes opens (expected)
 - "Wild Laugh" (not expected)
 - Message box (not expected)



Trojan Example

- ❑ How does freeMusic.mp3 trojan work?
- ❑ This “mp3” is an application, not data



- ❑ This trojan is harmless, but...
- ❑ ...could have done anything user could do
 - Delete files, download files, launch apps, etc.

Botnet

- ❑ Botnet: a “network” of infected machines
- ❑ Infected machines are “bots”
 - Victim is unaware of infection (stealthy)
- ❑ Botmaster controls botnet
 - Generally, using IRC
 - P2P botnet architectures exist
- ❑ Botnets used for...
 - Spam, DoS attacks, keylogging, ID theft, etc.

Botnet Examples

- ❑ XtremBot
 - Similar bots: Agobot, Forbot, Phatbot
 - Highly modular, easily modified
 - Source code readily available (GPL license)
- ❑ UrXbot
 - Similar bots: SDBot, UrBot, Rbot
 - Less sophisticated than XtremBot type
- ❑ GT-Bots and mIRC-based bots
 - mIRC is common IRC client for Windows

More Botnet Examples

- ❑ Mariposa
 - Used to steal credit card info
 - Creator arrested in July 2010
- ❑ Conficker
 - Estimated 10M infected hosts (2009)
- ❑ Kraken
 - Largest as of 2008 (400,000 infections)
- ❑ Srizbi
 - For spam, one of largest as of 2008

Ransomware

- ❑ Recently, ransomware very popular
- ❑ Ransomware encrypts important info
 - Key may be provided if ransom paid
 - Sometimes key works, sometimes not...
- ❑ To pay or not to pay?
 - Recent case, reportedly paid \$5M
 - In 2020 total ransom estimated \$350M
 - Payments usually in cryptocurrency

CryptoLocker Ransomware

- ❑ CryptoLocker (2013, Russian origin)
 - Email with attached “pdf” (actually an exe)
- ❑ When “pdf” launched...
 - 2048-bit RSA key pair generated
 - Public key on victim machine, private key with the attacker, data encrypted
 - Private key needed to decrypt data
- ❑ Payment of \$400 to decrypt data
 - Could be paid using Bitcoin

CryptoLocker

- ❑ In 2014, "Operation Tovar"
 - Server used by CryptoLocker taken down
 - Private keys recovered
- ❑ How many people paid?
 - Estimates ranged from 0.3% to 41%
 - Appears that actual number was about 1.3%
- ❑ Estimated that bad guys made \$3M

Locky of Ransomware

- ❑ Locky (2016, Russian origin)
 - Email with MS Word document
 - Word doc contained malicious macros
 - User had to enable macros
 - Doc filled with gibberish, "Enable macro if data encoding is incorrect"
- ❑ 2048-bit RSA and 128-bit AES
 - 1 Bitcoin (\$10k at the time)

WannaCry Ransomware

- ❑ WannaCry (2017, North Korean)
 - Worm based on "EternalBlue" exploit
 - EternalBlue supposedly developed by NSA, not revealed to MS for 5 years
- ❑ Asked for \$300 to \$600
- ❑ No data recovered after paying
 - Yet, payments of more than \$130k
- ❑ Included a hard-coded "kill switch"

WannaCry

- ❑ Infected 200k computers in 150 countries
- ❑ Countries most affected
 - Russia, Ukraine, India, Taiwan
- ❑ Long list of significant organizations affected
 - Notably, caused major disruption to British NHS



Petya Ransomware

- ❑ Petya (2017, origin uncertain)
- ❑ NotPetya (2017, Russian)
 - More advanced variant of Petya using EternalBlue
 - Targeted at Ukraine, but went worldwide
 - Blamed on Russian military
 - Not possible to recover from, even if ransom (\$300) was paid
- ❑ Disruptions caused \$10B in losses

Signature Detection

- ❑ A **signature** may be a string of bits in exe
 - Might also use wildcards, hash values, etc.
- ❑ For example, W32/Beast virus has signature
83EB 0274 EB0E 740A 81EB 0301 0000
 - That is, this string of bits appears in virus
- ❑ We can search for this signature in all files
- ❑ If string found, have we found W32/Beast?
 - Not necessarily —string could be in normal code
 - At random, chance is only $1/2^{112}$
 - But software is not random...

Signature Detection

❑ Advantages

- Effective on "ordinary" malware
- Minimal burden for users/administrators

❑ Disadvantages

- Signature file can be large (10s of thousands)...
- ...making scanning slow
- Signature files must be kept up to date
- ***Cannot detect unknown viruses***
- Cannot detect some advanced types of malware

❑ The most popular detection method

Change Detection

- ❑ Viruses must live somewhere
- ❑ If you detect a file has changed, it might have been infected
- ❑ How to detect changes?
 - Hash files and (securely) store hash values
 - Periodically re-compute hashes and compare
 - If hash 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 fall back on signature detection

Anomaly Detection

- ❑ Monitor system for anything “unusual” or “virus-like” or “potentially malicious” or ...
- ❑ Examples of anomalous things
 - Files change in some unexpected way
 - System misbehaves in some way
 - Unexpected network activity
 - Unexpected file access, etc., etc., etc., etc.
- ❑ But, we must first define “normal”
 - And normal can (and must) change over time

Anomaly Detection

- ❑ Advantages
 - Chance of detecting unknown malware
- ❑ Disadvantages
 - No proven track record
 - Trudy can make abnormal look normal (go slow)
 - Must be combined with another method (e.g., signature detection)
- ❑ Also popular in intrusion detection (IDS)
- ❑ Difficult unsolved (unsolvable?) problem
 - This sounds like a job for AI...

Machine Learning Based Malware Detection

- ❑ Essentially, higher-level “signature”
- ❑ Enables detection of many variants (entire “family”) with one model
- ❑ Many different features considered
- ❑ Many different machine learning and deep learning techniques used
- ❑ See next couple of slides...

ML Features

- ❑ Static feature (no execution required)
 - Opcodes, byte histogram, byte n-grams, PE header, entropy, graph structures, ...
 - Static features easier to collect
- ❑ Dynamic features (execute/emulate)
 - API calls, opcodes, graph structures, etc.
 - Dynamic features are harder for attackers to obfuscate

ML/DL Techniques

- ❑ Classic ML techniques
 - Hidden Markov models (HMM)
 - Support vector machines (SVM)
 - Random forest, k-nearest neighbor
- ❑ Neural networking techniques
 - Multilayer perceptron (basic neural net)
 - Convolutional neural networks (CNN)
 - Many, many, many others

Future of Malware

- ❑ Trends
 - Encrypted, polymorphic, metamorphic
 - Recently, botnets and ransomware
- ❑ The future is bright for malware!
 - Good news for the bad guys...
 - ...bad news for the good guys
- ❑ Future of malware detection?
 - ML/DL based techniques

Encrypted Viruses

- ❑ Virus writers know **signature detection** used
- ❑ So, how to evade signature detection?
- ❑ Encrypting the virus is a good approach
 - Ciphertext looks like random bits
 - Different key, then different “random” bits
 - So, different copies have no common signature
- ❑ Encryption often used in viruses today

Encrypted Viruses

- ❑ How to detect encrypted viruses?
- ❑ Scan for the decryptor code
 - More-or-less standard signature detection
 - But may be more false alarms
- ❑ Why not encrypt the decryptor code?
 - Then encrypt the decryptor of the decryptor (and so on...)
- ❑ Encryption of limited value to virus writers

Polymorphic Malware

❑ Polymorphic worm

- Body of worm is encrypted
- Decryptor code is "mutated" (or "morphed")
- Trying to hide decryptor signature
- Like an encrypted worm on steroids...

Q: How to detect?

A: Emulation —let the code decrypt itself

- Slow, and anti-emulation is possible

Metamorphic Malware

- ❑ A metamorphic worm mutates before infecting a new system
 - Sometimes called “body polymorphic”
- ❑ Such a worm can, in principle, evade signature-based detection
- ❑ Mutated worm must function the same
 - And be “different enough” to avoid detection
- ❑ Detection is a difficult research problem

Metamorphic Worm

- ❑ One approach to metamorphic replication...
 - The worm is disassembled
 - Worm then stripped to a base form
 - Random variations inserted into code (permute the code, insert dead code, etc., etc.)
 - Assemble the resulting code
- ❑ Result is a worm with same functionality as original, but different signature

Computer Infections

- ❑ Analogies are made between computer viruses/worms and biological diseases
- ❑ There are differences
 - Computer infections are much quicker
 - Ability to intervene in computer outbreak is more limited (vaccination?)
 - Bio disease models often not applicable
 - "Distance" almost meaningless on Internet
- ❑ But there are some similarities...

Computer Infections

- ❑ Cyber “diseases” vs biological diseases
- ❑ One similarity
 - In nature, too few susceptible individuals and disease will die out
 - In the Internet, too few susceptible systems and worm might fail to take hold
- ❑ One difference
 - In nature, diseases attack more-or-less at random
 - Cyber attackers select most “desirable” targets
 - Cyber attacks are more focused and damaging
- ❑ Mobile devices an interesting hybrid case

Future Malware Detection?

- ❑ Malware today far outnumber "goodware"
 - Metamorphic copies of existing malware
 - Many virus toolkits available
 - Trudy can recycle old viruses, new signatures
- ❑ So, may be better to "detect" good code
 - If code not on approved list, assume it's bad
 - That is, use **whitelist** instead of **blacklist**

Miscellaneous Software-Based Attacks

Miscellaneous Attacks

- ❑ Numerous attacks involve software
- ❑ We'll discuss a few issues that do not fit into previous categories
 - Salami attack
 - Linearization attack
 - Time bomb
 - Can you ever trust software?

Salami Attack

- ❑ What is Salami attack?
 - Programmer “slices off” small amounts of money
 - Slices are hard for victim to detect
- ❑ Example
 - Bank calculates interest on accounts
 - Programmer “slices off” any fraction of a cent and puts it in his own account
 - No customer notices missing partial cent
 - Bank may not notice any problem
 - Over time, programmer makes lots of money!

Salami Attack

- ❑ Such attacks are possible for insiders
- ❑ Do salami attacks actually occur?
 - Or is it just Office Space folklore?
- ❑ Programmer added a few cents to every employee payroll tax withholding
 - But money credited to programmer's tax
 - Programmer got a big tax refund!
- ❑ Rent-a-car franchise in Florida inflated gas tank capacity to overcharge customers

Salami Attacks

- ❑ Employee reprogrammed Taco Bell cash register: \$2.99 item registered as \$0.01
 - Employee pocketed \$2.98 on each such item
 - A large “slice” of salami!
- ❑ In LA, four men installed computer chip that overstated amount of gas pumped
 - Customers complained when they had to pay for more gas than tank could hold
 - Hard to detect since chip programmed to give correct amount when 5 or 10 gallons purchased
 - Inspector usually asked for 5 or 10 gallons

Linearization Attack

- ❑ Program checks for serial number S123N456
- ❑ For efficiency, check made one character at a time
- ❑ Can attacker take advantage of this?

```
#include <stdio.h>

int main(int argc, const char *argv[])
{
    int i;
    char serial[9]="S123N456\n";

    for(i = 0; i < 8; ++i)
    {
        if(argv[1][i] != serial[i]) break;
    }
    if(i == 8)
    {
        printf("\nSerial number is correct!\n\n");
    }
}
```

Linearization Attack

- ❑ Correct number takes longer than incorrect
- ❑ Trudy tries all 1st characters
 - Find that S takes longest
- ❑ Then she guesses all 2nd characters: S*
- Finds S1 takes longest
- ❑ And so on...
- ❑ Trudy can recover one character at a time!
 - Same principle as used in lock picking

Linearization Attack

- ❑ What is the advantage to attacking serial number one character at a time?
- ❑ Suppose serial number is 8 characters and each has 128 possible values
 - Then $128^8 = 2^{56}$ possible serial numbers
 - Attacker would guess the serial number in about 2^{55} tries —a lot of work!
 - Using the linearization attack, the work is about $8 * (128/2) = 2^9$ which is easy

Linearization Attack

- ❑ A real-world linearization attack
- ❑ TENEX, an ancient OS (timeshare)
 - Passwords checked one character at a time
 - Careful timing was *not* necessary, instead...
 - ...could arrange for a "page fault" when next unknown character guessed correctly
 - Page fault register was user accessible
- ❑ Attack was very easy in practice

Time Bomb

- ❑ In 1986 Donald Gene Burleson told employer to stop withholding taxes from his paycheck
- ❑ His company refused
- ❑ He planned to sue his company
 - He used company time to prepare legal docs
 - Company found out and fired him
- ❑ Burleson had been working on malware...
 - After being fired, his software "time bomb" deleted important company data

Time Bomb

- ❑ Company was reluctant to pursue the case
- ❑ So Burleson sued company for back pay!
 - Then company finally sued Burleson
- ❑ In 1988 Burleson fined \$11,800
 - Case took years to prosecute...
 - Cost company thousands of dollars...
 - Resulted in a slap on the wrist for attacker
- ❑ One of the first computer crime cases
- ❑ Many cases since follow a similar pattern
 - Companies reluctant to prosecute

Trusting Software

- ❑ Can you ever trust software?
 - See [Reflections on Trusting Trust](#)
- ❑ Consider the following thought experiment
- ❑ Suppose C compiler has a virus
 - When compiling login program, virus creates backdoor (account with known password)
 - When recompiling the C compiler, virus incorporates itself into new C compiler
- ❑ Difficult to get rid of this virus!

Trusting Software

- ❑ Suppose you notice something is wrong
- ❑ So you start over from scratch
- ❑ First, you recompile the C compiler
- ❑ Then you recompile the OS
 - Including login program...
 - You have not gotten rid of the problem!
- ❑ In the real world
 - Attackers try to hide viruses in virus scanner
 - Imagine damage that would be done by attack on virus signature updates

Chapter 12:

Insecurity in Software

Every time I write about the impossibility of effectively protecting digital files on a general-purpose computer, I get responses from people decrying the death of copyright. “How will authors and artists get paid for their work?” they ask me. Truth be told, I don’t know. I feel rather like the physicist who just explained relativity to a group of would-be interstellar travelers, only to be asked: “How do you expect us to get to the stars, then?”

I’m sorry, but I don’t know that, either.

—Bruce Schneier

So much time and so little to do! Strike that. Reverse it. Thank you.

—Willy Wonka

Software Engineering (SRE)

SRE

- ❑ **Software Reverse Engineering**
 - Also known as Reverse Code Engineering (RCE)
 - Or simply “reversing”
- ❑ Can be used for **good**...
 - Understand malware
 - Understand legacy code
- ❑ ...or **not-so-good**
 - Remove usage restrictions from software
 - Find and exploit flaws in software
 - Cheat at games, etc.

SRE

- ❑ We assume...
 - Reverse engineer is an attacker
 - Attacker only has exe (no source code)
 - No bytecode (i.e., not Java, .Net, etc.)
- ❑ Attacker might want to
 - Understand the software
 - Modify ("patch") the software
- ❑ SRE usually focused on Windows
 - So we focus on Windows

SRE Tools

- ❑ Disassembler
 - Converts exe to assembly (as best it can)
 - Cannot always disassemble 100% correctly
 - In general, not possible to re-assemble disassembly into working executable
- ❑ Debugger
 - Must step thru code to completely understand it
 - Labor intensive —lack of useful tools
- ❑ Hex Editor
 - To **patch** (modify) exe file
- ❑ Process Monitor, VMware, etc.

Specific Tools

- ❑ **IDA Pro** —good disassembler/debugger
 - Costs a few hundred dollars (free version exists)
 - Converts binary to assembly (as best it can)
- ❑ **OllyDbg** —high-quality shareware debugger
 - Includes a good disassembler
- ❑ **Hex editor** —to view/modify bits of exe
 - UltraEdit is good —freeware
 - HIEW —useful for patching exe
- ❑ **Process Monitor** —freeware

Why is Debugger Needed?

- ❑ Disassembly gives **static** results
 - Good overview of program logic
 - User must “mentally execute” program
 - Difficult to jump to specific place in the code
- ❑ Debugging is **dynamic**
 - Can set break points
 - Can treat complex code as “black box”
 - And code not always disassembled correctly
- ❑ Disassembly **and** debugging **both** required for any serious SRE task

SRE Necessary Skills

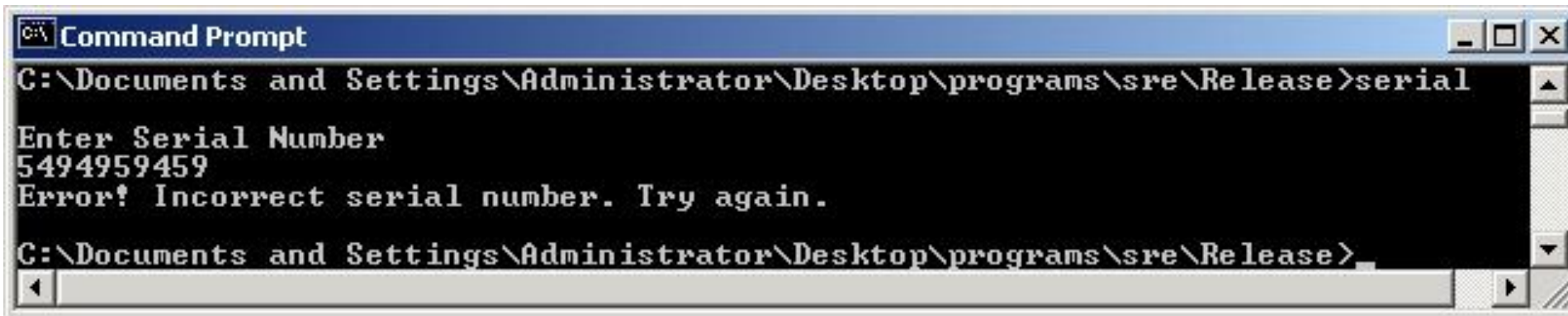
- ❑ Working knowledge of target assembly code
- ❑ Experience with the tools
 - IDA Pro —sophisticated and complex
 - OllyDbg —good choice for this class
- ❑ Knowledge of Windows **Portable Executable** (PE) file format
- ❑ Boundless patience and optimism
 - SRE is a tedious and labor-intensive process
 - Often, like finding a needle in a haystack

SRE Example

- ❑ We consider a simple example
- ❑ This example only requires disassembly (IDA Pro used here) and hex editor
 - Trudy disassembles to understand code
 - Trudy also wants to patch (modify) the code
- ❑ For most real-world code, would also need a debugger (e.g., OllyDbg)

SRE Example

- ❑ Program requires serial number
- ❑ But Trudy doesn't know the serial number...



```
Command Prompt
C:\Documents and Settings\Administrator\Desktop\programs\sre\Release>serial
Enter Serial Number
5494959459
Error! Incorrect serial number. Try again.
C:\Documents and Settings\Administrator\Desktop\programs\sre\Release>
```

- ❑ Can Trudy get serial number from exe?

SRE Example

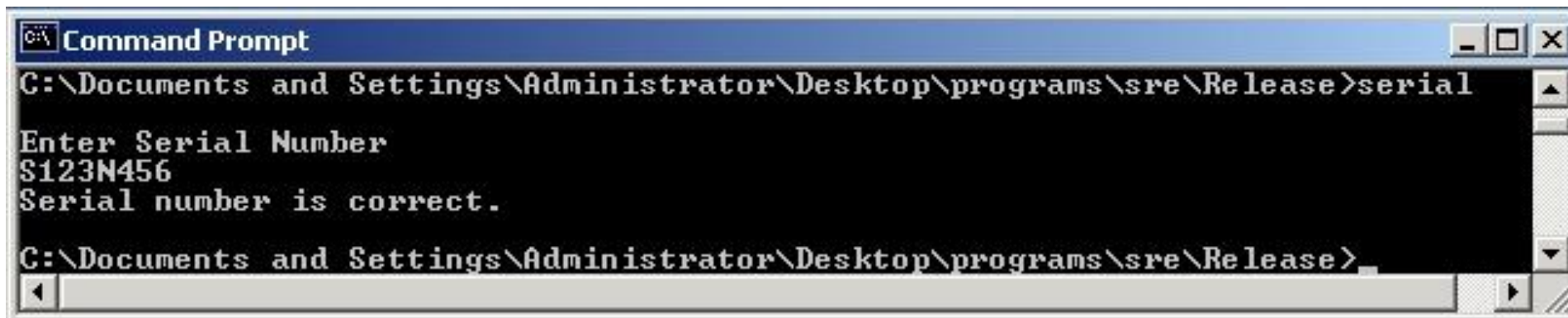
❑ IDA Pro disassembly

```
.text:00401003      push    offset aEnterSerialNum ; "\nEnter Serial Number\n"
.text:00401008      call    sub_4010AF
.text:0040100D      lea     eax, [esp+18h+var_14]
.text:00401011      push    eax
.text:00401012      push    offset aS              ; "%5"
.text:00401017      call    sub_401098
.text:0040101C      push    8
.text:0040101E      lea     ecx, [esp+24h+var_14]
.text:00401022      push    offset aS123n456 ; "S123N456"
.text:00401027      push    ecx
.text:00401028      call    sub_401060
.text:0040102D      add     esp, 18h
.text:00401030      test    eax, eax
.text:00401032      jz      short loc_401045
.text:00401034      push    offset aErrorIncorrect ; "Error! Incorrect serial number."
.text:00401039      call    sub_4010AF
```

❑ Looks like serial number is S123N456

SRE Example

- ❑ Try the serial number S123N456



```
Command Prompt
C:\Documents and Settings\Administrator\Desktop\programs\sre\Release>serial

Enter Serial Number
S123N456
Serial number is correct.

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

- ❑ It works!
- ❑ Can Trudy do "better"?

SRE Example

□ Again, IDA Pro disassembly

```
.text:00401003      push    offset aEnterSerialNum ; "\nEnter Serial Number\n"
.text:00401008      call    sub_4010AF
.text:0040100D      lea     eax, [esp+18h+var_14]
.text:00401011      push    eax
.text:00401012      push    offset aS              ; "%s"
.text:00401017      call    sub_401098
.text:0040101C      push    8
.text:0040101E      lea     ecx, [esp+24h+var_14]
.text:00401022      push    offset aS123n456 ; "S123N456"
.text:00401027      push    ecx
.text:00401028      call    sub_401060
.text:0040102D      add     esp, 18h
.text:00401030      test    eax, eax
.text:00401032      jz      short loc_401045
.text:00401034      push    offset aErrorIncorrect ; "Error! Incorrect serial number."
.text:00401039      call    sub_4010AF
```

□ And hex view...

```
.text:00401010      04 50 68 84 80 40 00 E8-7C 00 00 00 6A 08 8D 4C
.text:00401020      24 10 68 78 80 40 00 51-E8 33 00 00 00 83 C4 18
.text:00401030      85 C0 74 11 68 4C 80 40-00 E8 71 00 00 00 83 C4
.text:00401040      04 83 C4 14 C3 68 30 80-40 00 E8 60 00 00 00 83
```

SRE Example

```
.text:00401003      push    offset aEnterSerialNum ; "\nEnter Serial Number\n"
.text:00401008      call     sub_4010AF
.text:0040100D      lea      eax, [esp+18h+var_14]
.text:00401011      push    eax
.text:00401012      push    offset aS          ; "%S"
.text:00401017      call     sub_401098
.text:0040101C      push    8
.text:0040101E      lea      ecx, [esp+24h+var_14]
.text:00401022      push    offset aS123n456 ; "S123N456"
.text:00401027      push    ecx
.text:00401028      call     sub_401060
.text:0040102D      add     esp, 18h
.text:00401030      test    eax, eax
.text:00401032      jz       short loc_401045
.text:00401034      push    offset aErrorIncorrect ; "Error! Incorrect serial number."
.text:00401039      call     sub_4010AF
```

- ❑ “test eax,eax” is AND of eax with itself
 - So, zero flag set only if eax is 0
 - If test yields 0, then jz is true
- ❑ Trudy wants jz to always be true
- ❑ Can Trudy patch exe so jz always holds?

SRE Example

- Can Trudy patch exe so that jz always true?

```

.text:00401003      push    offset aEnterSerialNum ; "\nEnter Serial Number\n"
.text:00401008      call    sub_4010AF
.text:0040100D      lea     eax, [esp+18h+var_14]
.text:00401011      push    eax
.text:00401012      push    offset aS          ; "%5"
.text:00401017      call    sub_401098
.text:0040101C      push    8
.text:0040101E      lea     ecx, [esp+24h+var_14]
.text:00401022      push    offset aS123n456 ; "S123N456"
.text:00401027      push    ecx
.text:00401028      call    sub_401060
.text:0040102D      add     esp, 18h
.text:00401030      test  eax, eax
.text:00401032      jz      short loc_401045 ← jz always true!!!
.text:00401034      push    offset aErrorIncorrect ; "Error! Incorrect serial number."
.text:00401039      call    sub_4010AF

```

Assembly		Hex
test	eax, eax	85 C0 ...
xor	eax, eax	33 C0 ...

SRE Example

- Can edit serial.exe with hex editor

serial.exe

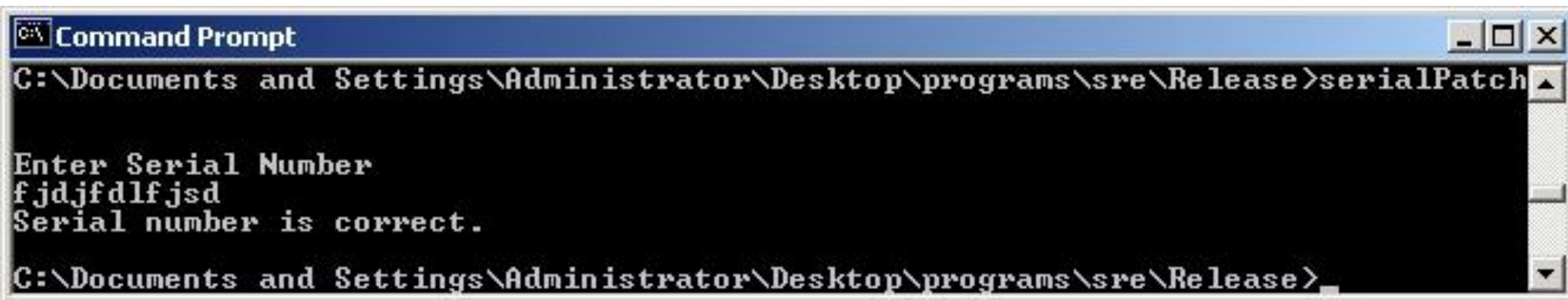
00001010h:	04	50	68	84	80	40	00	E8	7C	00	00	00	6A	08	8D	4C
00001020h:	24	10	68	78	80	40	00	51	E8	33	00	00	00	83	C4	18
00001030h:	85	CD	74	11	68	4C	80	40	00	E8	71	00	00	00	83	C4
00001040h:	04	83	C4	14	C3	68	30	80	40	00	E8	60	00	00	00	83
00001050h:	C4	04	83	C4	14	C3	90	90	90	90	90	90	90	90	90	90

serialPatch.exe

00001010h:	04	50	68	84	80	40	00	E8	7C	00	00	00	6A	08	8D	4C
00001020h:	24	10	68	78	80	40	00	51	E8	33	00	00	00	83	C4	18
00001030h:	33	CD	74	11	68	4C	80	40	00	E8	71	00	00	00	83	C4
00001040h:	04	83	C4	14	C3	68	30	80	40	00	E8	60	00	00	00	83
00001050h:	C4	04	83	C4	14	C3	90	90	90	90	90	90	90	90	90	90

- Save as serialPatch.exe

SRE Example



```
Command Prompt
C:\Documents and Settings\Administrator\Desktop\programs\sre\Release>serialPatch

Enter Serial Number
fjdjfdlfjsd
Serial number is correct.

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

- ❑ **Any** "serial number" now works!
- ❑ Very convenient for Trudy

SRE Example

❑ Back to IDA Pro disassembly...

serial.exe

```
.text:00401003  
.text:00401008  
.text:0040100D  
.text:00401011  
.text:00401012  
.text:00401017  
.text:0040101C  
.text:0040101E  
.text:00401022  
.text:00401027  
.text:00401028  
.text:0040102D  
.text:00401030  
.text:00401032  
.text:00401034  
.text:00401039
```

```
push    offset aEnterSerialNum ; "\nEnter Serial Number\n"  
call    sub_4010AF  
lea     eax, [esp+18h+var_14]  
push    eax  
push    offset aS              ; "%5"  
call    sub_401098  
push    8  
lea     ecx, [esp+24h+var_14]  
push    offset aS123n456 ; "S123N456"  
push    ecx  
call    sub_401060  
add     esp, 18h  
test    eax, eax  
jz      short loc_401045  
push    offset aErrorIncorrect ; "Error! Incorrect serial number."  
call    sub_4010AF
```

serialPatch.exe

```
.text:00401003  
.text:00401008  
.text:0040100D  
.text:00401011  
.text:00401012  
.text:00401017  
.text:0040101C  
.text:0040101E  
.text:00401022  
.text:00401027  
.text:00401028  
.text:0040102D  
.text:00401030  
.text:00401032  
.text:00401034  
.text:00401039
```

```
push    offset aEnterSerialNum ; "\nEnter Serial Number\n"  
call    sub_4010AF  
lea     eax, [esp+18h+var_14]  
push    eax  
push    offset aS              ; "%5"  
call    sub_401098  
push    8  
lea     ecx, [esp+24h+var_14]  
push    offset aS123n456 ; "S123N456"  
push    ecx  
call    sub_401060  
add     esp, 18h  
xor     eax, eax  
jz      short loc_401045  
push    offset aErrorIncorrect ; "Error! Incorrect serial number."  
call    sub_4010AF
```

SRE Attack Mitigation

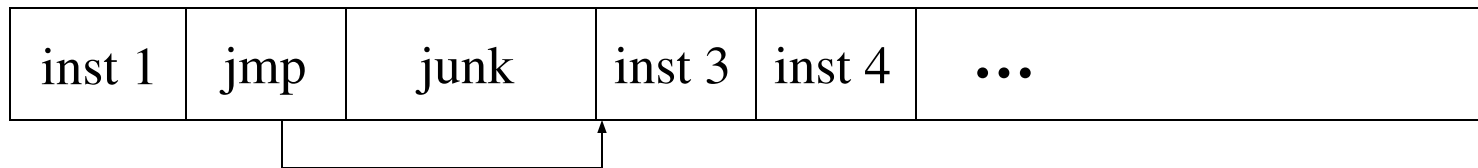
- ❑ **Impossible** to prevent SRE on open systems
- ❑ Can we make such attacks more difficult?
- ❑ Anti-disassembly techniques
 - To confuse static view of code
- ❑ Anti-debugging techniques
 - To confuse dynamic view of code
- ❑ Tamper-resistance
 - Code checks itself to detect tampering
- ❑ Code obfuscation
 - Make code more difficult to understand

Anti-disassembly

- ❑ Anti-disassembly methods include
 - Encrypted or “packed” object code
 - False disassembly
 - Self-modifying code
 - Many other techniques
- ❑ Encryption **prevents** disassembly
 - But need plaintext decryptor to decrypt code!
 - Same problem as with polymorphic viruses

Anti-disassembly Example

- Suppose actual code instructions are



- What a “dumb” disassembler sees



- This is example of “false disassembly”
- Persistent attacker will figure it out

Anti-debugging

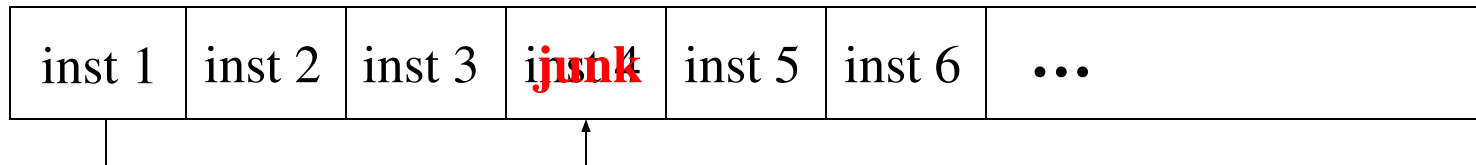
- ❑ IsDebuggerPresent() function
- ❑ Can also monitor for
 - Use of debug registers
 - Inserted breakpoints
- ❑ Debuggers don't handle **threads** well
 - Interacting threads may confuse debugger...
 - ...and therefore, confuse attacker
- ❑ Many other debugger-unfriendly tricks
 - See next slide for one example

Anti-debugger Example

inst 1	inst 2	inst 3	inst 4	inst 5	inst 6	...
--------	--------	--------	--------	--------	--------	-----

- ❑ Suppose when **program** gets inst 1, it pre-fetches inst 2, inst 3, and inst 4
 - This is done to increase efficiency
- ❑ Suppose when **debugger** executes inst 1, it does **not** pre-fetch instructions
- ❑ Can we use this difference to confuse the debugger?

Anti-debugger Example



- ❑ Suppose inst 1 **overwrites** inst 4 in memory
- ❑ Then program (without debugger) will be OK since it fetched inst 4 at same time as inst 1
- ❑ Debugger will be confused when it reaches **junk** where inst 4 is supposed to be
- ❑ Problem if this segment of code executed more than once!
 - Also, self-modifying code is platform-dependent
- ❑ Again, clever attacker can figure this out

Tamper-resistance

- ❑ Goal is to make patching more difficult
- ❑ Code can **hash** parts of itself
- ❑ If tampering occurs, hash check fails
- ❑ Research has shown, can get good coverage of code with small performance penalty
- ❑ But don't want all checks to look similar
 - Or else easy for attacker to remove checks
- ❑ This approach sometimes called "guards"

Code Obfuscation

- ❑ Goal is to make code hard to understand
 - Opposite of good software engineering
 - Spaghetti code is a good example
- ❑ Much research into more robust obfuscation
 - Example: **opaque predicate**

```
int x,y  
:  
if((x-y)*(x-y) > (x*x-2*x*y+y*y)){...}
```

 - The if() conditional is always false
- ❑ Attacker wastes time analyzing dead code

Code Obfuscation

- ❑ Code obfuscation sometimes promoted as a powerful security technique
- ❑ Diffie and Hellman's original idea for public key crypto was based on code obfuscation
 - But did not prove to be useful approach
- ❑ It has been shown that obfuscation probably cannot provide strong, crypto-like security
 - [On the \(im\)possibility of obfuscating programs](#)
- ❑ Obfuscation might still have practical uses
 - Even if it can never be as strong as crypto

Authentication Example

- ❑ Software used to determine authentication
- ❑ Ultimately, authentication is 1-bit decision
 - Regardless of method used (pwd, biometric, ...)
 - Somewhere in authentication software, a single bit determines success/failure
- ❑ If Trudy can find this bit, she can force authentication to always succeed
- ❑ Obfuscation makes it more difficult for attacker to find this all-important bit

Obfuscation

- ❑ Obfuscation forces attacker to analyze larger amounts of code
- ❑ Method could be combined with
 - Anti-disassembly techniques
 - Anti-debugging techniques
 - Code tamper-checking
- ❑ All of these increase work/pain for attacker
- ❑ But a persistent attacker can ultimately win

Software Cloning

- ❑ Suppose we write a piece of software
- ❑ We then distribute an identical copy (or clone) to each customers
- ❑ If an attack is found on one copy, the same attack works on all copies
- ❑ This approach has no resistance to “break once, break everywhere” (BOBE)
- ❑ This is the usual situation in software development

Metamorphic Software

- ❑ Metamorphism sometimes used in malware
- ❑ Can metamorphism also be used for good?
- ❑ Suppose we write a piece of software
- ❑ Each copy we distribute is different
 - This is an example of metamorphic software
- ❑ Two levels of metamorphism are possible
 - All instances are functionally distinct (only possible in certain application)
 - All instances are functionally identical but differ internally (always possible)
 - We consider the latter case

Metamorphic Software

- ❑ If we distribute N copies of cloned software
 - One successful attack breaks all N
- ❑ If we distribute N metamorphic copies, where each of N instances is functionally identical, but they differ internally...
 - An attack on one instance does not necessarily work against other instances
 - In the best case, N times as much work is required to break all N instances

Metamorphic Software

- ❑ We cannot prevent SRE attacks
- ❑ The best we can hope for is BOBE resistance
- ❑ Metamorphism can improve BOBE resistance
- ❑ Consider the analogy to genetic diversity
 - If all plants in a field are genetically identical, one disease can rapidly kill **all** of the plants
 - If the plants in a field are genetically diverse, one disease can only kill **some** of the plants

Cloning vs Metamorphism

- ❑ Spse our software has a buffer overflow
- ❑ **Cloned** software
 - Same buffer overflow attack will work against **all** cloned copies of the software
- ❑ **Metamorphic** software
 - Unique instances —all are functionally the same, but they differ in internal structure
 - Buffer overflow likely exists in all instances
 - But a specific buffer overflow attack will only work against **some** instances
 - Buffer overflow attacks are delicate!

Metamorphic Software

- ❑ Metamorphic software is intriguing concept
- ❑ But raises concerns regarding...
 - Software development, upgrades, etc.
- ❑ Metamorphism does not prevent SRE, but could make it infeasible on a large scale
- ❑ Metamorphism might be a practical tool for increasing BOBE resistance
- ❑ Metamorphism currently used in malware
- ❑ So, metamorphism is not just for evil!

Digital Rights Management

Digital Rights Management

- ❑ DRM is a good example of limitations of doing security in software
- ❑ We'll discuss
 - What is DRM?
 - A PDF document protection system
 - DRM for streaming media
 - DRM in P2P application
 - DRM within an enterprise

What is DRM?

- ❑ “Remote control” problem
 - Distribute digital content
 - Retain some control on its use, **after delivery**
- ❑ **Digital book** example
 - Digital book sold online could have huge market
 - But might only sell 1 copy!
 - Trivial to make perfect digital copies
 - A fundamental change from pre-digital era
- ❑ Similar comments for digital music, video, etc.

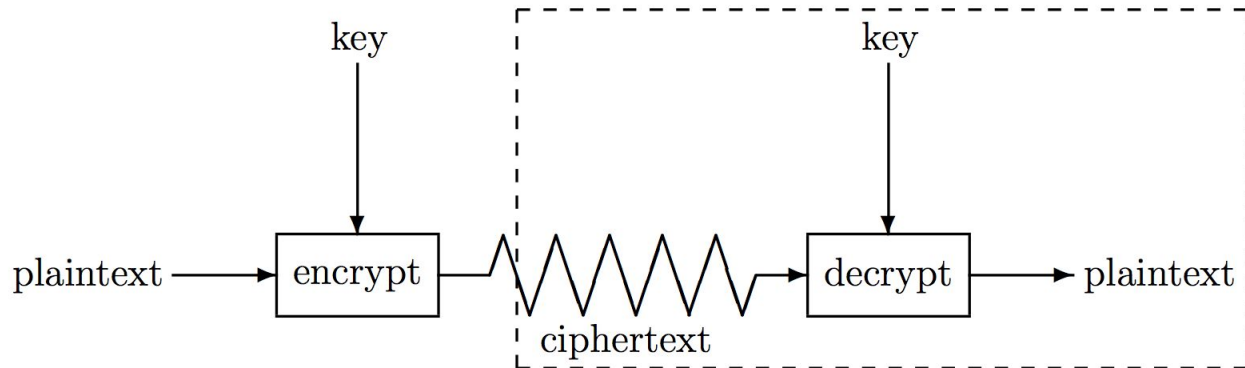
Persistent Protection

- ❑ “Persistent protection” is the fundamental problem in DRM
 - How to enforce restrictions on use of content **after** delivery?
- ❑ Examples of such restrictions
 - No copying
 - Limited number of reads/plays
 - Time limits
 - No forwarding, etc.

What Can be Done?

- ❑ The honor system?
 - Example: Stephen King's, *The Plant*
- ❑ Give up?
 - Internet sales? Regulatory compliance? etc.
- ❑ Lame software-based DRM?
 - The standard DRM system today
- ❑ Better software-based DRM?
 - MediaSnap's goal
- ❑ Tamper-resistant hardware?
 - Closed systems: Game Cube, etc.
 - Open systems: TCG/NGSCB for PCs

Is Crypto the Answer?



- ❑ Attacker's goal is to recover the **key**
- ❑ In standard crypto scenario, attacker has
 - Ciphertext, some plaintext, side-channel info, etc.
- ❑ In DRM scenario, attacker has
 - Everything in the box (at least)
- ❑ Crypto was not designed for this problem!

Is Crypto the Answer?

- ❑ But crypto is necessary
 - To securely deliver the bits
 - To prevent trivial attacks
- ❑ Then attacker will not try to directly attack crypto
- ❑ Attacker will try to find keys in software
 - DRM is “hide and seek” with keys in software!

Current State of DRM

- ❑ At best, **security by obscurity**
 - A derogatory term in security
- ❑ Secret designs
 - In violation of **Kerckhoffs Principle**
- ❑ Over-reliance on crypto
 - "Whoever thinks his problem can be solved using cryptography, doesn't understand his problem and doesn't understand cryptography."
—Attributed by Roger Needham and Butler Lampson to each other

DRM Limitations

- ❑ The **analog hole**
 - When content is rendered, it can be captured in analog form
 - DRM **cannot** prevent such an attack
- ❑ **Human nature** matters
 - Absolute DRM security is impossible
 - Want something that “works” in practice
 - What works depends on context
- ❑ DRM is not strictly a technical problem!

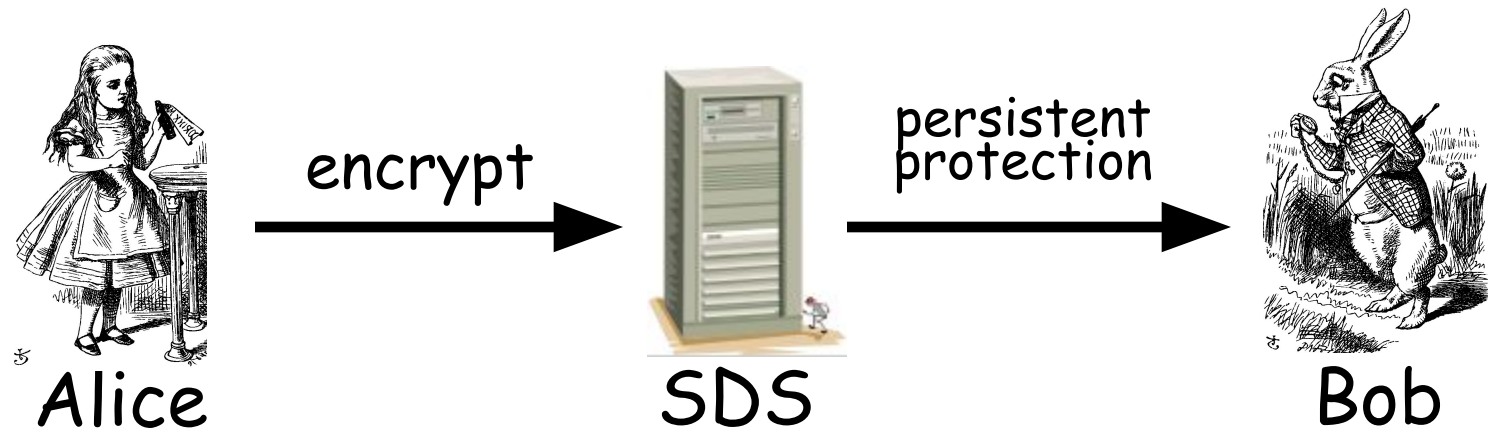
Software-based DRM

- ❑ Strong software-based DRM is impossible
- ❑ Why?
 - We can't really hide a secret in software
 - We cannot prevent SRE
 - User with full admin privilege can eventually break any anti-SRE protection
- ❑ Bottom line: **The** killer attack on software-based DRM is SRE

DRM for PDF Documents

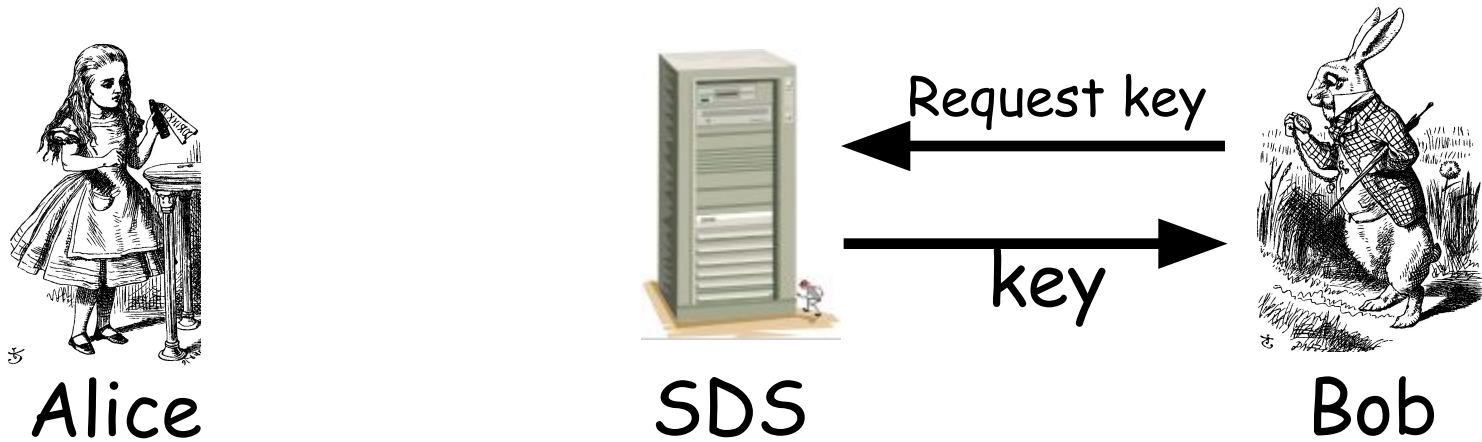
- ❑ Based on design of MediaSnap, Inc., a small Silicon Valley startup company
- ❑ Developed a DRM system
 - Designed to protect PDF documents
- ❑ Two parts to the system
 - Server — Secure Document Server (SDS)
 - Client — PDF Reader “plugin” software

Protecting a Document



- ❑ Alice creates PDF document
- ❑ Document encrypted and sent to SDS
- ❑ SDS applies desired "persistent protection"
- ❑ Document sent to Bob

Accessing a Document

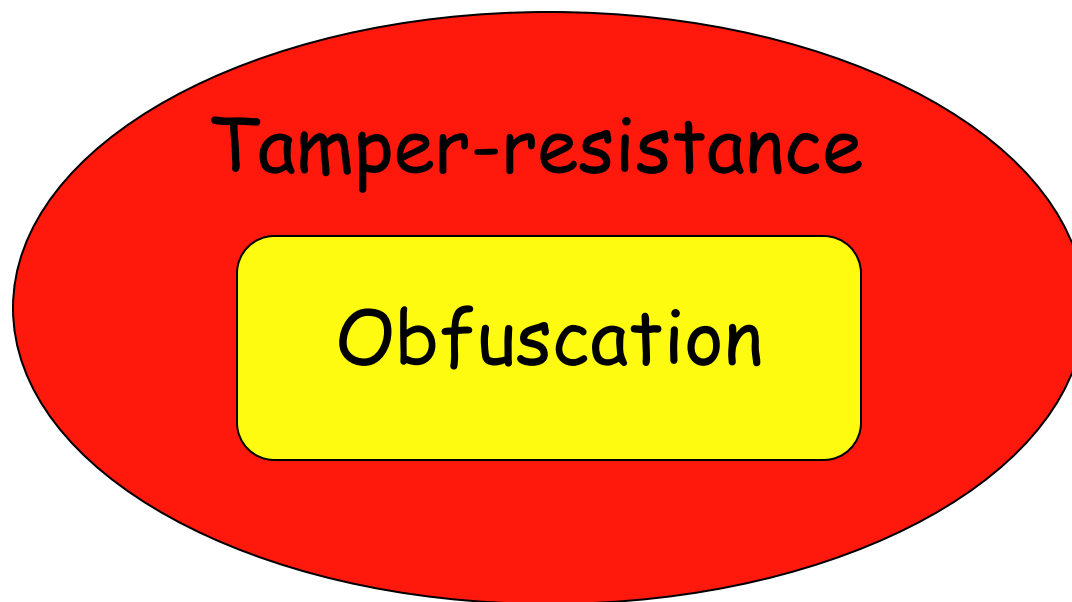


- ❑ Bob authenticates to SDS
- ❑ Bob requests key from SDS
- ❑ Bob can then access document, but only thru special DRM software

Security Issues

- ❑ Server side (SDS)
 - Protect keys, authentication data, etc.
 - Apply persistent protection
- ❑ Client side (PDF plugin)
 - Protect keys, authenticate user, etc.
 - Enforce persistent protection
- ❑ Remaining discussion concerns **client**

Security Overview



- ❑ A tamper-resistant outer layer
- ❑ Software obfuscation applied within

Tamper-Resistance



- ❑ Encrypted code will prevent static analysis of PDF plugin software
- ❑ Anti-debugging to prevent dynamic analysis of PDF plugin software
- ❑ These two designed to protect each other
- ❑ But the persistent attacker will get thru!

Obfuscation

- ❑ Obfuscation can be used for
 - Key management
 - Authentication
 - Caching (keys and authentication info)
 - Encryption and “scrambling”
 - Key parts (data and/or code)
 - Multiple keys/key parts
- ❑ Obfuscation can only slow the attacker
- ❑ The persistent attacker still wins!

Other Security Features

- ❑ Code tamper checking (hashing)
 - To validate all code executing on system
- ❑ Anti-screen capture
 - To prevent obvious attack on digital documents
- ❑ Watermarking
 - In theory, can trace stolen content
 - In practice, of limited value
- ❑ Metamorphism (or individualization)
 - For BOBE-resistance

Security Not Implemented

- ❑ More general code obfuscation
- ❑ Code “fragilization”
 - Code that hash checks itself
 - Tampering should cause code to break
- ❑ OS cannot be trusted
 - How to protect against “bad” OS?
 - Not an easy problem!

DRM for Streaming Media

- ❑ Stream digital content over Internet
 - Usually audio or video
 - Viewed in real time
- ❑ Want to charge money for the content
- ❑ Can we protect content from capture?
 - So content can't be redistributed
 - We want to make money!

Attacks on Streaming Media

- ❑ Spoof the stream between endpoints
- ❑ Man in the middle
- ❑ Replay and/or redistribute data
- ❑ **Capture the plaintext**
 - This is the threat we are concerned with
 - Must prevent malicious software from capturing plaintext stream at client end

Design Features

- ❑ Scrambling algorithms
 - Encryption-like algorithms
 - Many distinct algorithms available
 - A strong form of metamorphism!
- ❑ Negotiation of scrambling algorithm
 - Server and client must both know the algorithm
- ❑ Decryption at receiver end
 - To remove the strong encryption
- ❑ De-scrambling in device driver
 - De-scramble just prior to rendering

Scrambling Algorithms

- ❑ Server has a large set of scrambling algorithms
 - Suppose N of these numbered 1 thru N
- ❑ Each client has a subset of algorithms
 - For example: LIST = {12,45,2,37,23,31}
- ❑ The LIST is stored on client, encrypted with server's key: $E(\text{LIST}, K_{\text{server}})$

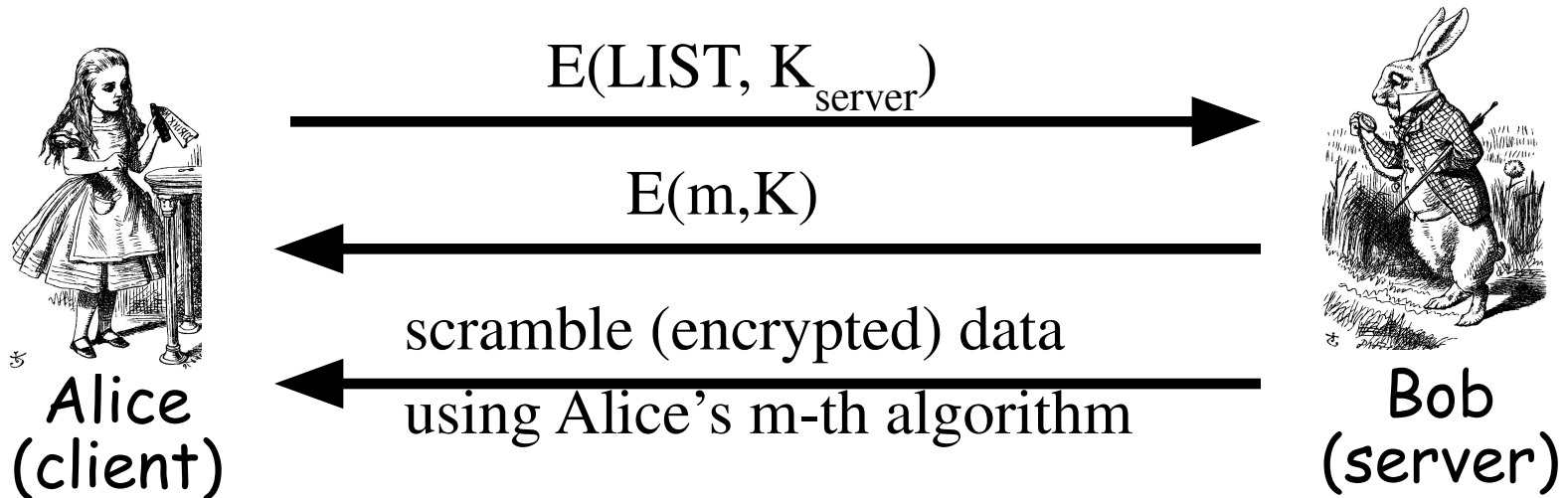
Server-side Scrambling

- ❑ On server side



- ❑ Server must scramble data with an algorithm the client supports
- ❑ Client must send server list of algorithms it supports
- ❑ Server must securely communicate algorithm choice to client

Select Scrambling Algorithm



- ❑ The key K is a session key
- ❑ The LIST is unreadable by client
 - Reminiscent of Kerberos TGT

Client-side De-scrambling

- On client side



- Try to keep plaintext away from potential attacker
- "Proprietary" device driver
 - Scrambling algorithms "baked in"
 - Able to de-scramble at last moment

Why Scrambling?

- ❑ **Metamorphism** deeply embedded in system
- ❑ If a scrambling algorithm is known to be broken, server will not choose it
- ❑ If client has too many broken algorithms, server can force software upgrade
- ❑ Proprietary algorithm harder for SRE
- ❑ We cannot trust crypto strength of proprietary algorithms, so we also encrypt

Why Metamorphism?

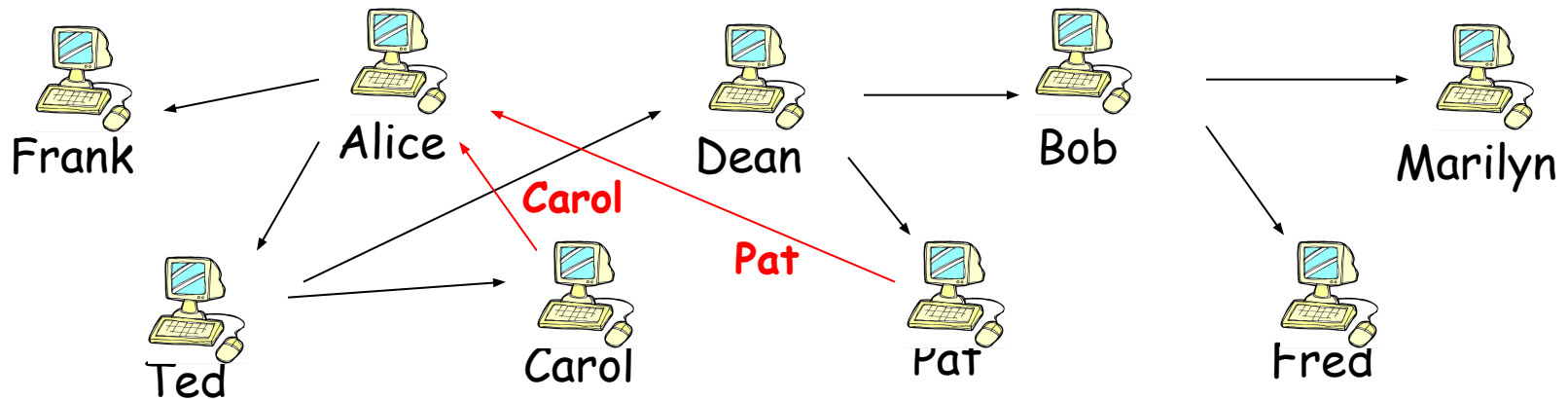
- ❑ The most serious threat is **SRE**
- ❑ Attacker does not need to reverse engineer any standard crypto algorithm
 - Attacker only needs to find the key
- ❑ Reverse engineering a scrambling algorithm may be difficult
- ❑ This is just **security by obscurity**
- ❑ But appears to help with BOBE-resistance

DRM for a P2P Application

- ❑ Today, much digital content is delivered via peer-to-peer (P2P) networks
 - P2P networks contain lots of pirated music
- ❑ Is it possible to get people to pay for digital content on such P2P networks?
- ❑ How can this possibly work?
- ❑ A peer offering service (POS) is one idea

P2P File Sharing: Query

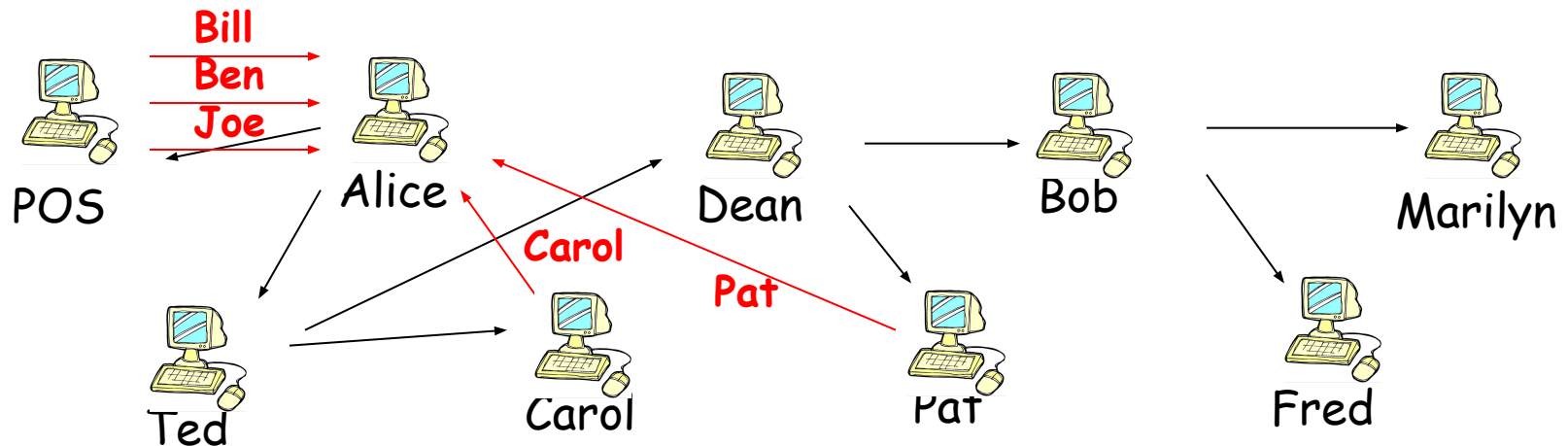
- Suppose Alice requests "Hey Jude"
- **Black** arrows: query flooding
- **Red** arrows: positive responses



- Alice can select from: **Carol, Pat**

P2P File Sharing with POS

- Suppose Alice requests "Hey Jude"
- Black arrow: query
- Red arrow: positive response



- Alice selects from: **Bill**, **Ben**, **Carol**, **Joe**, **Pat**
- **Bill**, **Ben**, and **Joe** have legal content!

POS

- ❑ Bill, Ben and Joe must appear normal to Alice
- ❑ If “victim” (Alice) clicks POS response
 - DRM protected (legal) content downloaded
 - **Then** small payment required to play
- ❑ Alice can choose not to pay
 - But then she must download again
 - Is it worth the hassle to avoid paying small fee?
 - POS content can also offer extras

POS Conclusions

- ❑ A very clever idea!
- ❑ Piggybacking on existing P2P networks
- ❑ Weak DRM works very well here
 - Pirated content already exists
 - DRM only needs to be more hassle to break than the hassle of clicking and waiting
- ❑ Current state of POS?
 - Very little interest from the music industry
 - Considerable interest from the “adult” industry

DRM in the Enterprise

- ❑ Why enterprise DRM?
- ❑ Health Insurance Portability and Accountability Act (HIPAA)
 - Medical records must be protected
 - Fines of up to \$10,000 “per incident”
- ❑ Sarbanes-Oxley Act (SOA)
 - Must preserve documents of interest to SEC
- ❑ DRM-like protections needed by corporations for **regulatory compliance**

What's Different in Enterprise DRM?

- ❑ Technically, similar to e-commerce
- ❑ But motivation for DRM is different
 - Regulatory compliance
 - To satisfy a legal requirement
 - Not to make money —to avoid losing money!
- ❑ Human dimension is completely different
 - Legal threats are far more plausible
- ❑ Legally, corporation is OK provided an **active attack** on DRM is required

Enterprise DRM

- ❑ Moderate DRM security is sufficient
- ❑ **Policy management issues**
 - Easy to set policies for groups, roles, etc.
 - Yet policies must be flexible
- ❑ **Authentication issues**
 - Must interface with existing system
 - Must prevent network authentication spoofing (authenticate the authentication server)
- ❑ Enterprise DRM is a solvable problem!

DRM Failures

- ❑ Many examples of DRM failures
 - One system defeated by a felt-tip pen
 - One defeated by holding down shift key
 - Secure Digital Music Initiative (SDMI) completely broken before it was finished
 - Adobe eBooks
 - Microsoft MS-DRM (version 2)
 - Many, many others!

DRM Conclusions

- ❑ DRM nicely illustrates limitations of doing security in software
- ❑ Software in a hostile environment is extremely vulnerable to attack
- ❑ Protection options are very limited
- ❑ Attacker has enormous advantage
- ❑ Tamper-resistant hardware and a trusted OS can make a difference
 - We'll discuss this more later: TCG/NGSCB

Secure Software Development

Penetrate and Patch

- ❑ Usual approach to software development
 - Develop product as quickly as possible
 - Release it without adequate testing
 - Patch the code as flaws are discovered
- ❑ In security, this is “penetrate and patch”
 - A **bad** approach to software development
 - An **even worse** approach to secure software!

Why Penetrate and Patch?

- ❑ First to market advantage
 - First to market likely to become market leader
 - Market leader has huge advantage in software
 - Users find it safer to “follow the leader”
 - Boss won't complain if your system has a flaw, as long as everybody else has same flaw...
 - User can ask more people for support, etc.
- ❑ Sometimes called “network economics”

Why Penetrate and Patch?

- ❑ Secure software development is hard
 - Costly and time consuming development
 - Costly and time consuming testing
 - Cheaper to let customers do the work!
- ❑ No serious economic disincentive
 - Even if software flaw causes major losses, the software vendor is not liable
 - Is any other product sold this way?
 - Would it matter if vendors were legally liable?

Penetrate and Patch Fallacy

- ❑ **Fallacy:** If you keep patching software, eventually it will be secure
- ❑ Why is this a fallacy?
- ❑ Empirical evidence to the contrary
- ❑ Patches often add new flaws
- ❑ Software is a moving target: new versions, features, changing environment, new uses,...

Open vs Closed Source

- ❑ Open source software
 - The source code is available to user
 - For example, Linux
- ❑ Closed source
 - The source code is not available to user
 - For example, Windows
- ❑ What are the security implications?

Open Source Security

- ❑ Claimed advantages of open source is
 - **More eyeballs:** more people looking at the code should imply fewer flaws
 - A variant on Kerchoffs Principle
- ❑ Is this valid?
 - How many “eyeballs” looking for security flaws?
 - How many “eyeballs” focused on boring parts?
 - How many “eyeballs” belong to security experts?
 - Attackers can also look for flaws!
 - Evil coder might be able to insert a flaw

Open Source Security

- ❑ Open source example: wu-ftp
 - About 8,000 lines of code
 - A security-critical application
 - Was deployed and widely used
 - After 10 years, serious security flaws discovered!
- ❑ More generally, open source software has done little to reduce security flaws
- ❑ Why?
 - Open source follows penetrate and patch model!

Closed Source Security

- ❑ Claimed advantage of closed source
 - Security flaws not as visible to attacker
 - This is a form of “security by obscurity”
- ❑ Is this valid?
 - Many exploits do not require source code
 - Possible to analyze closed source code...
 - ...though it is a lot of work!
 - Is “security by obscurity” real security?

Open vs Closed Source

- ❑ Advocates of open source often cite the **Microsoft fallacy** which states
 1. Microsoft makes bad software
 2. Microsoft software is closed source
 3. Therefore all closed source software is bad
- ❑ Why is this a fallacy?
 - Not logically correct
 - More relevant is the fact that Microsoft follows the penetrate and patch model

Open vs Closed Source

- ❑ No obvious security advantage to either open or closed source
- ❑ More significant than open vs closed source is software development practices
- ❑ Both open and closed source follow the “penetrate and patch” model

Open vs Closed Source

- ❑ If there is no security difference, why is Microsoft software attacked so often?
 - Microsoft is a big target!
 - Attacker wants most “bang for the buck”
- ❑ Few exploits against Mac OS X
 - **Not** because OS X is inherently more secure
 - An OS X attack would do less damage
 - Would bring less “glory” to attacker
- ❑ Next, we consider the theoretical differences
 - [See this paper](#)

Security and Testing

- Can be shown that probability of a security failure after t units of testing is about
 $E = K/t$ where K is a constant
- This approximation holds over large range of t
- Then the “mean time between failures” is
 $MTBF = t/K$
- The good news: security improves with testing
- The bad news: security only improves **linearly** with testing!

Security and Testing

- ❑ The “mean time between failures” is approximately
$$\text{MTBF} = t/K$$
- ❑ To have 1,000,000 hours between security failures, must test 1,000,000 hours!
- ❑ Suppose **open source** project has $\text{MTBF} = t/K$
- ❑ If flaws in **closed source** are twice as hard to find, do we then have $\text{MTBF} = 2t/K$?
 - No! Testing not as effective $\text{MTBF} = 2(t/2)/K = t/K$
- ❑ The same result for open and closed source!

Security and Testing

- ❑ Closed source advocates might argue
 - Closed source has "open source" alpha testing, where flaws found at (higher) open source rate
 - Followed by closed source beta testing and use, giving attackers the (lower) closed source rate
 - Does this give closed source an advantage?
- ❑ Alpha testing is minor part of total testing
 - Recall, first to market advantage
 - Products rushed to market
- ❑ Probably no real advantage for closed source

Security and Testing

- ❑ No security difference between open and closed source?
- ❑ Provided that flaws are found “linearly”
- ❑ Is this valid?
 - Empirical results show security improves linearly with testing
 - Conventional wisdom is that this is the case for large and complex software systems

Security and Testing

- ❑ The fundamental problem
 - Good guys must find (almost) all flaws
 - Bad guy only needs 1 (exploitable) flaw
- ❑ Software reliability far more difficult in security than elsewhere
- ❑ How much more difficult?
 - See the next slide...

Security Testing: Do the Math

- ❑ Recall that $MTBF = t/K$
- ❑ Suppose 10^6 security flaws in some software
 - Say, Windows XP
- ❑ Suppose each bug has MTBF of 10^9 hours
- ❑ Expect to find 1 bug for every 10^3 hours testing
- ❑ Good guys spend 10^7 hours testing: **find 10^4 bugs**
 - Good guys have found 1% of all the bugs
- ❑ Trudy spends 10^3 hours of testing: **finds 1 bug**
- ❑ Chance good guys found Trudy's bug is only **1% !!!**

Software Development

- General software development model
 - Specify
 - Design
 - Implement
 - Test
 - Review
 - Document
 - Manage
 - Maintain



Secure Software Development

- ❑ Goal: move away from “penetrate and patch”
- ❑ Penetrate and patch will always exist
 - But if more care taken in development, then fewer and less severe flaws to patch
- ❑ Secure software development not easy
- ❑ Much more time and effort required thru entire development process
- ❑ Today, little economic incentive for this!

Secure Software Development

- ❑ We briefly discuss the following
 - Design
 - Hazard analysis
 - Peer review
 - Testing
 - Configuration management
 - Postmortem for mistakes

Design

- ❑ Careful initial design
- ❑ Try to avoid high-level errors
 - Such errors may be impossible to correct later
 - Certainly costly to correct these errors later
- ❑ Verify assumptions, protocols, etc.
- ❑ Usually informal approach is used
- ❑ Formal methods
 - Possible to rigorously **prove** design is correct
 - In practice, only works in simple cases

Hazard Analysis

- Hazard analysis (or threat modeling)
 - Develop hazard list
 - List of what ifs
 - Schneier's "attack tree"
- Many formal approaches
 - Hazard and operability studies (HAZOP)
 - Failure modes and effective analysis (FMEA)
 - Fault tree analysis (FTA)

Peer Review

- ❑ Three levels of peer review
 - Review (informal)
 - Walk-through (semi-formal)
 - Inspection (formal)
- ❑ Each level of review is important
- ❑ Much evidence that peer review is effective
- ❑ Although programmers might not like it!

Levels of Testing

- ❑ Module testing —test each small section of code
- ❑ Component testing —test combinations of a few modules
- ❑ Unit testing —combine several components for testing
- ❑ Integration testing —put everything together and test

Types of Testing

- ❑ Function testing —verify that system functions as it is supposed to
- ❑ Performance testing —other requirements such as speed, resource use, etc.
- ❑ Acceptance testing —customer involved
- ❑ Installation testing —test at install time
- ❑ Regression testing —test after any change

Other Testing Issues

- ❑ Active fault detection
 - Don't wait for system to fail
 - Actively try to make it fail —attackers will!
- ❑ Fault injection
 - Insert faults into the process
 - Even if no obvious way for such a fault to occur
- ❑ Bug injection
 - Insert bugs into code
 - See how many of injected bugs are found
 - Can use this to estimate number of bugs
 - Assumes injected bugs similar to unknown bugs

Testing Case History

- ❑ In one system with 184,000 lines of code
- ❑ Flaws found
 - 17.3% inspecting system design
 - 19.1% inspecting component design
 - 15.1% code inspection
 - 29.4% integration testing
 - 16.6% system and regression testing
- ❑ Conclusion: must do many kinds of testing
 - Overlapping testing is necessary
 - Provides a form of “defense in depth”

Security Testing: The Bottom Line

- ❑ **Security testing** is far more demanding than non-security testing
- ❑ Non-security testing —does system do what it is supposed to?
- ❑ Security testing —does system do what it is supposed to **and nothing more?**
- ❑ Usually impossible to do exhaustive testing
- ❑ How much testing is enough?

Security Testing: The Bottom Line

- ❑ How much testing is enough?
- ❑ Recall $MTBF = t/K$
- ❑ Seems to imply testing is nearly hopeless!
- ❑ But there is some hope...
 - If we eliminate an entire class of flaws then statistical model breaks down
 - For example, if a single test (or a few tests) find all buffer overflows

Configuration Issues

- ❑ Types of changes
 - Minor changes —maintain daily functioning
 - Adaptive changes —modifications
 - Perfective changes —improvements
 - Preventive changes —no loss of performance
- ❑ Any change can introduce new flaws!

Postmortem

- ❑ After fixing any security flaw...
- ❑ Carefully analyze the flaw
- ❑ To learn from a mistake
 - Mistake must be analyzed and understood
 - Must make effort to avoid repeating mistake
- ❑ In security, **always** learn more when things go wrong than when they go right
- ❑ Postmortem may be the most under-used tool in all of security engineering!

Software Security

- ❑ First to market advantage
 - Also known as “network economics”
 - Security suffers as a result
 - Little economic incentive for secure software!
- ❑ **Penetrate and patch**
 - Fix code as security flaws are found
 - Fix can result in worse problems
 - Mostly done **after** code delivered
- ❑ Proper development can reduce flaws
 - But costly and time-consuming

Software and Security

- ❑ Even with best development practices, security flaws will still exist
- ❑ Absolute security is (almost) never possible
- ❑ So, it is not surprising that absolute software security is impossible
- ❑ The goal is to minimize and manage risks of software flaws
- ❑ Do not expect dramatic improvements in consumer software security anytime soon!

Chapter 13:

Operating Systems and Security

UNIX is basically a simple operating system,
but you have to be a genius to understand the simplicity.

— Dennis Ritchie

And it is a mark of prudence never to trust wholly
in those things which have once deceived us.

—Rene Descartes

OS and Security

- ❑ OSs are large, complex programs
 - Many bugs in any such program
 - We have seen that bugs can be security threats
- ❑ Here we are concerned with security provided by OS
 - Not concerned with threat of bad OS software
- ❑ Concerned with OS as security **enforcer**
- ❑ In this section we only scratch the surface

OS Security Challenges

- ❑ Modern OS is **multi-user** and **multi-tasking**
- ❑ OS must deal with
 - Memory
 - I/O devices (disk, printer, etc.)
 - Programs, threads
 - Network issues
 - Data, etc.
- ❑ OS must protect processes from other processes and users from other users
 - Whether accidental or malicious

OS Security Functions

- ❑ Memory protection
 - Protect memory from users/processes
- ❑ File protection
 - Protect user and system resources
- ❑ Authentication
 - Determines and enforce authentication results
- ❑ Authorization
 - Determine and enforces access control

Memory Protection

- ❑ Fundamental problem
 - How to keep users/processes separate?
- ❑ Separation
 - Physical separation —separate devices
 - Temporal separation —one at a time
 - Logical separation —sandboxing, etc.
 - Cryptographic separation —make information unintelligible to outsider
 - Or any combination of the above

Memory Protection

- ❑ **Fence** —users cannot cross a specified address
 - Static fence —fixed size OS
 - Dynamic fence —fence register
- ❑ Base/bounds register —lower and upper address limit
- ❑ Assumes contiguous space



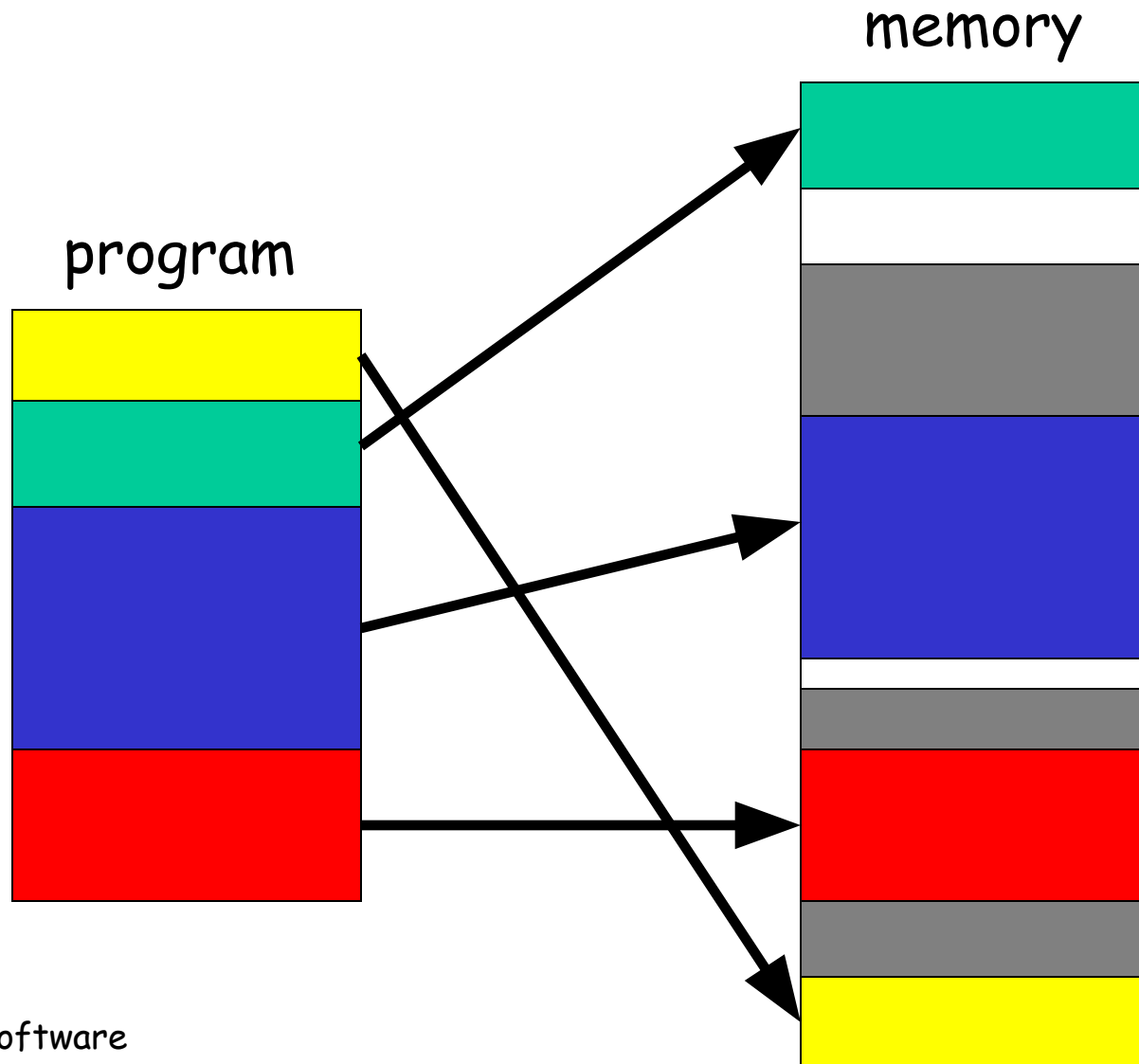
Memory Protection

- ❑ Tagging — specify protection of each address
 - + Extremely fine-grained protection
 - High overhead — can be reduced by tagging sections instead of individual addresses
 - Compatibility
- ❑ More common is segmentation and/or paging
 - Protection is not as flexible
 - But much more efficient

Segmentation

- ❑ Divide memory into logical units, such as
 - Single procedure
 - Data in one array, etc.
- ❑ Can enforce different access restrictions on different segments
- ❑ Any segment can be placed in any memory location (if location is large enough)
- ❑ OS keeps track of actual locations

Segmentation



Segmentation

- ❑ OS can place segments anywhere
- ❑ OS keeps track of segment locations as `<segment,offset>`
- ❑ Segments can be moved in memory
- ❑ Segments can move out of memory
- ❑ All address references go thru OS

Segmentation Advantages

- ❑ Every address reference can be checked
 - Possible to achieve **complete mediation**
- ❑ Different protection can be applied to different segments
- ❑ Users can share access to segments
- ❑ Specific users can be restricted to specific segments

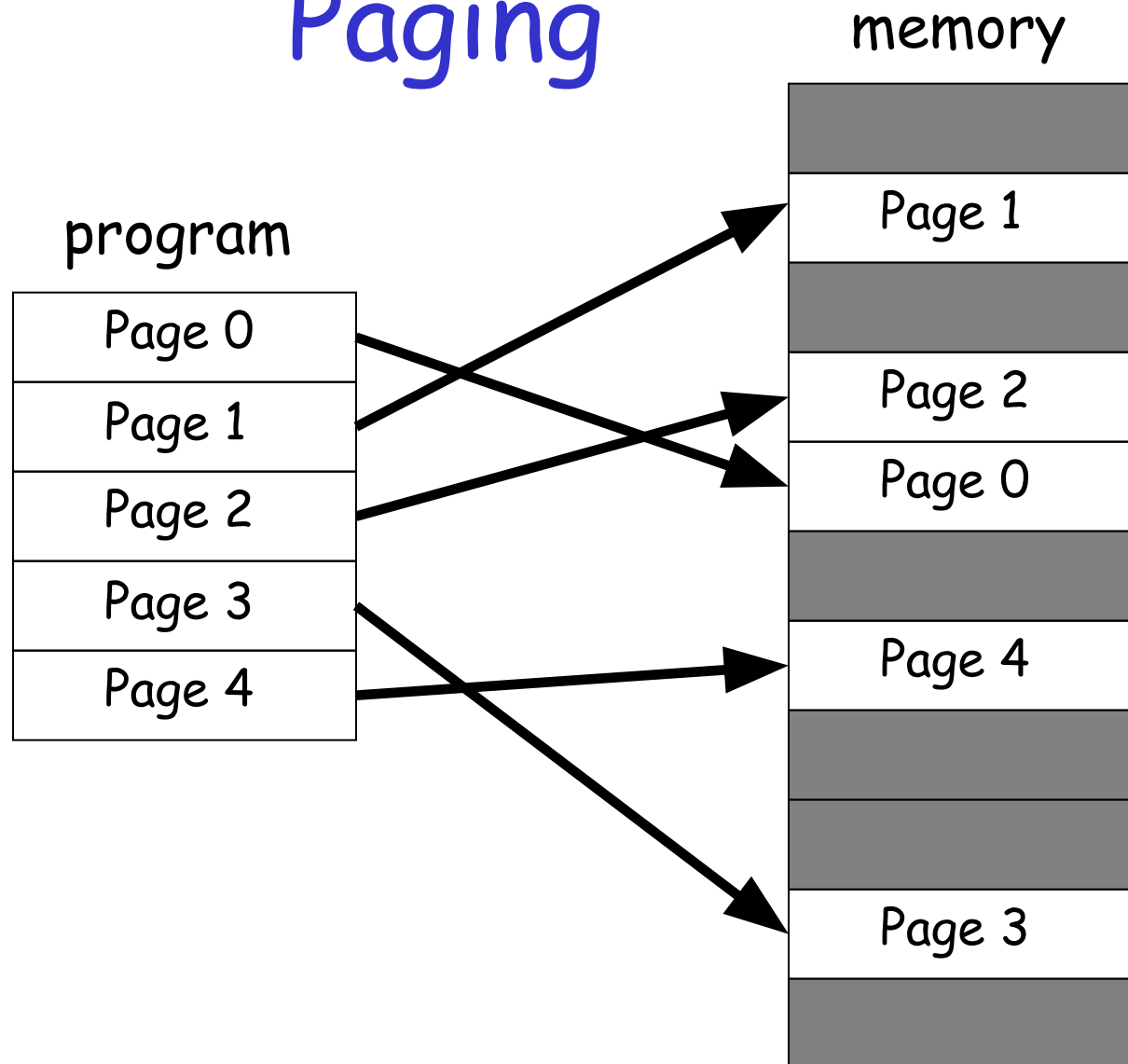
Segmentation Disadvantages

- ❑ How to reference $\langle \text{segment}, \text{offset} \rangle$?
 - OS must know segment **size** to verify access is within segment
 - But some segments can grow during execution (for example, dynamic memory allocation)
 - OS must keep track of **variable** segment sizes
- ❑ Memory fragmentation is also a problem
 - Compacting memory changes tables
- ❑ A lot of work for the OS
- ❑ More complex \Rightarrow more chance for mistakes

Paging

- ❑ Like segmentation, but fixed-size segments
- ❑ Access via <page,offset>
- ❑ Plusses and minuses
 - + Avoids fragmentation, improved efficiency
 - + OS need not keep track of variable segment sizes
 - No logical unity to pages
 - What protection to apply to a given page?

Paging



Other OS Security Functions

- ❑ OS must enforce access control
- ❑ Authentication
 - Passwords, biometrics
 - Single sign-on, etc.
- ❑ Authorization
 - ACL
 - Capabilities
- ❑ These topics discussed previously
- ❑ OS is an attractive target for attack!

Trusted Operating System

Trusted Operating System

- ❑ An OS is **trusted** if we rely on it for
 - Memory protection
 - File protection
 - Authentication
 - Authorization
- ❑ Every OS does these things
- ❑ But if a trusted OS fails to provide these, our security fails

Trust vs Security

- ❑ **Trust** implies reliance
 - ❑ Trust is binary
 - ❑ Ideally, only trust secure systems
 - ❑ All trust relationships should be explicit
 - ❑ **Security** is a judgment of effectiveness
 - ❑ Judge based on specified policy
 - ❑ Security depends on trust relationships
-
- ❑ Note: Some authors use different terminology!

Trusted Systems

- ❑ **Trust** implies reliance
- ❑ A trusted system is relied on for security
- ❑ An untrusted system is not relied on for security
- ❑ If all untrusted systems are compromised, your security is unaffected
- ❑ Ironically, **only a trusted system can break your security!**

Trusted OS

- ❑ OS mediates interactions between subjects (users) and objects (resources)
- ❑ Trusted OS must decide
 - Which objects to protect and how
 - Which subjects are allowed to do what

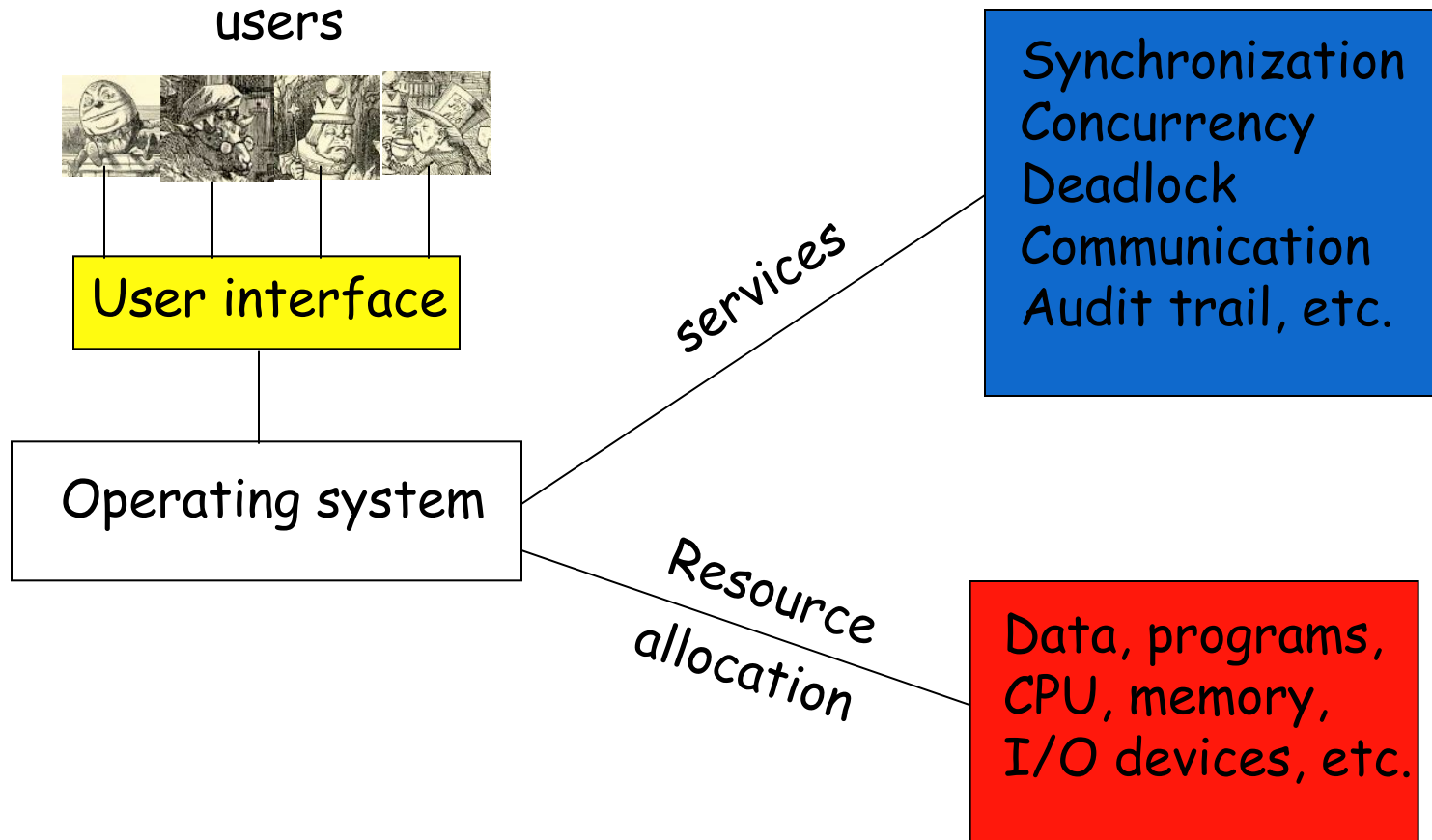
General Security Principles

- ❑ Least privilege —like “low watermark”
- ❑ Simplicity
- ❑ Open design (Kerchoffs Principle)
- ❑ Complete mediation
- ❑ White listing (preferable to black listing)
- ❑ Separation
- ❑ Ease of use
- ❑ But commercial OSs emphasize **features**
 - Results in complexity and poor security

OS Security

- ❑ Any OS must provide some degree of
 - Authentication
 - Authorization (users, devices and data)
 - Memory protection
 - Sharing
 - Fairness
 - Inter-process communication/synchronization
 - OS protection

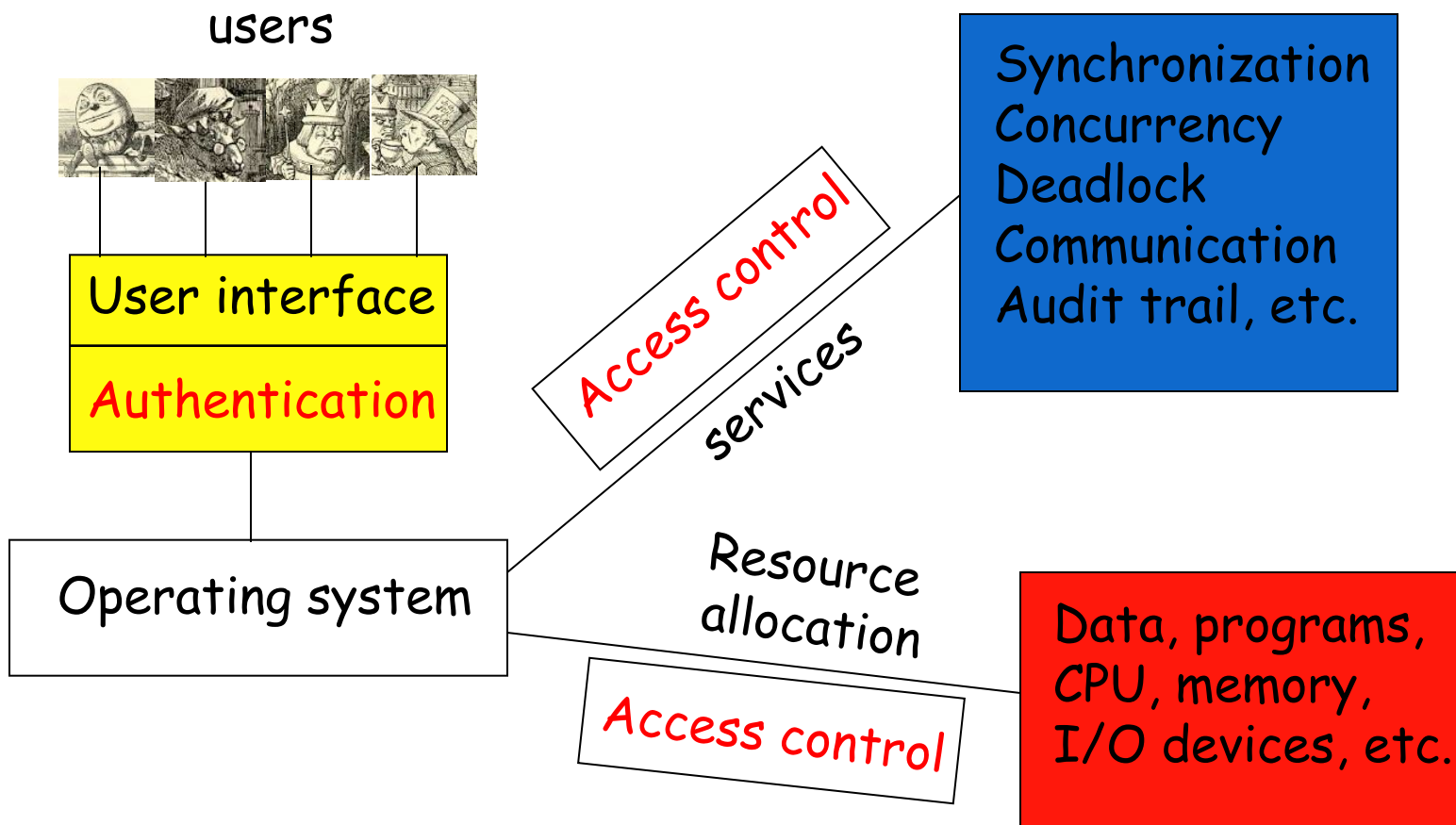
OS Services



Trusted OS

- ❑ A trusted OS also provides some or all of
 - User authentication/authorization
 - Mandatory access control (**MAC**)
 - Discretionary access control (**DAC**)
 - Object reuse protection
 - Complete mediation —access control
 - Trusted path
 - Audit/logs

Trusted OS Services



MAC and DAC

- ❑ Mandatory Access Control (MAC)
 - Access not controlled by owner of object
 - Example: User does not decide who holds a TOP SECRET clearance
- ❑ Discretionary Access Control (DAC)
 - Owner of object determines access
 - Example: UNIX/Windows file protection
- ❑ If DAC and MAC both apply, MAC wins

Object Reuse Protection

- ❑ OS must prevent leaking of info
- ❑ Example
 - User creates a file
 - Space allocated on disk
 - But same space previously used
 - “Leftover” bits could leak information
 - Magnetic remanence is a related issue

Trusted Path

- ❑ Suppose you type in your password
 - What happens to the password?
- ❑ Depends on the software!
- ❑ How can you be sure software is not evil?
- ❑ Trusted path problem:

“I don't know how to be confident even of a digital signature I make on my own PC, and I've worked in security for over fifteen years. Checking all of the software in the critical path between the display and the signature software is way beyond my patience.”

—Ross Anderson

Audit

- ❑ System should log security-related events
- ❑ Necessary for postmortem
- ❑ What to log?
 - Everything? Who (or what) will look at it?
 - Don't want to overwhelm administrator
 - Needle in haystack problem
- ❑ Should we log incorrect passwords?
 - "Almost" passwords in log file?
- ❑ Logging is not a trivial matter

Security Kernel

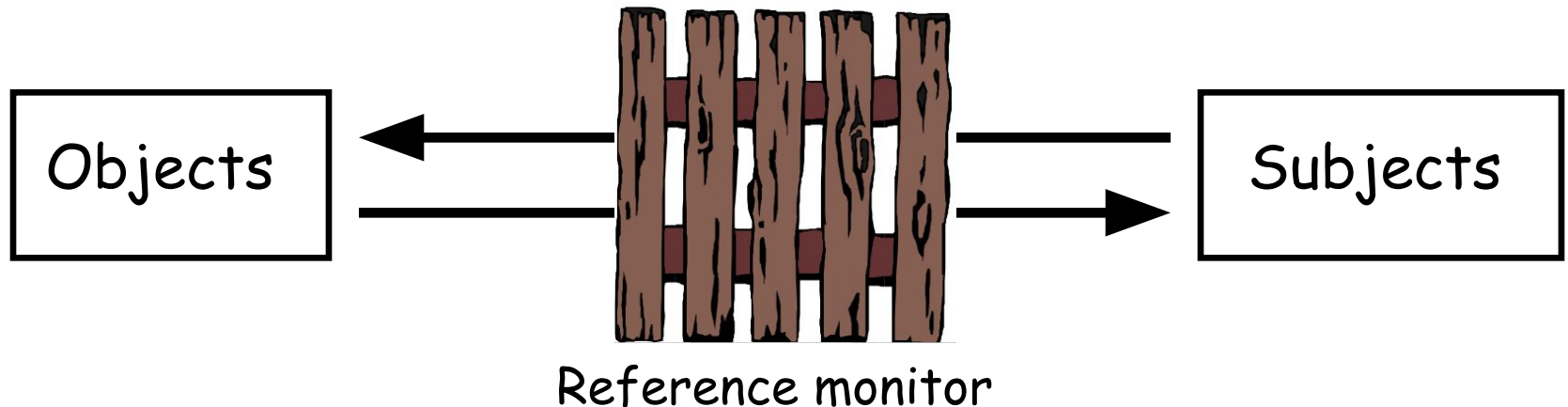
- ❑ **Kernel** is the lowest-level part of the OS
- ❑ Kernel is responsible for
 - Synchronization
 - Inter-process communication
 - Message passing
 - Interrupt handling
- ❑ The **security kernel** is the part of the kernel that deals with security
- ❑ Security kernel contained within the kernel

Security Kernel

- ❑ Why have a security kernel?
- ❑ All accesses go thru kernel
 - Ideal place for access control
- ❑ Security-critical functions in one location
 - Easier to analyze and test
 - Easier to modify
- ❑ More difficult for attacker to get in “below” security functions

Reference Monitor

- ❑ The part of the security kernel that deals with access control
 - Mediates access of subjects to objects
 - Tamper-resistant
 - Analyzable (small, simple, etc.)



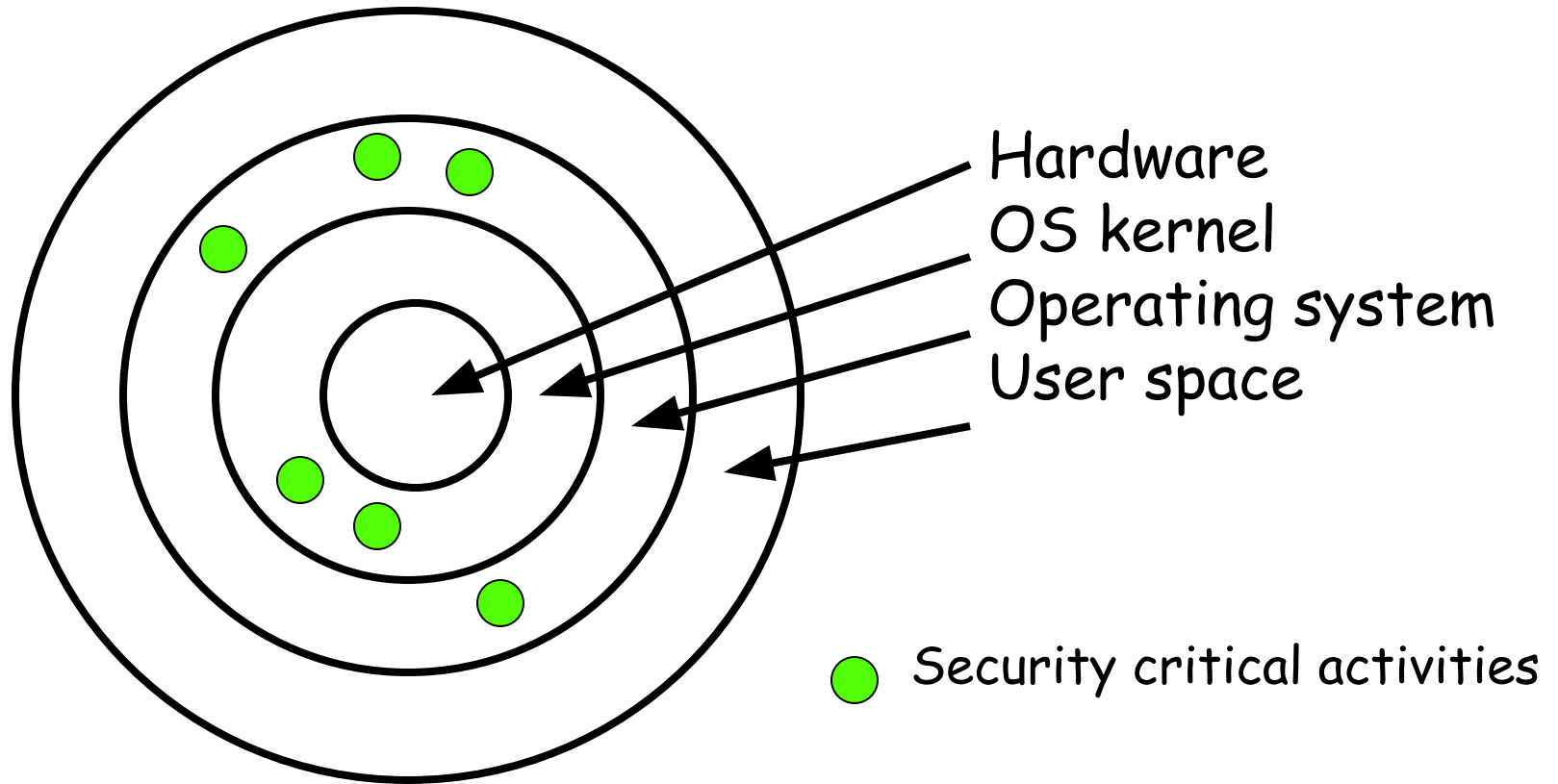
Trusted Computing Base

- ❑ **TCB** —everything in the OS that we rely on to enforce security
- ❑ If everything outside TCB is subverted, trusted OS would still be trusted
- ❑ TCB protects users from each other
 - Context switching between users
 - Shared processes
 - Memory protection for users
 - I/O operations, etc.

TCB Implementation

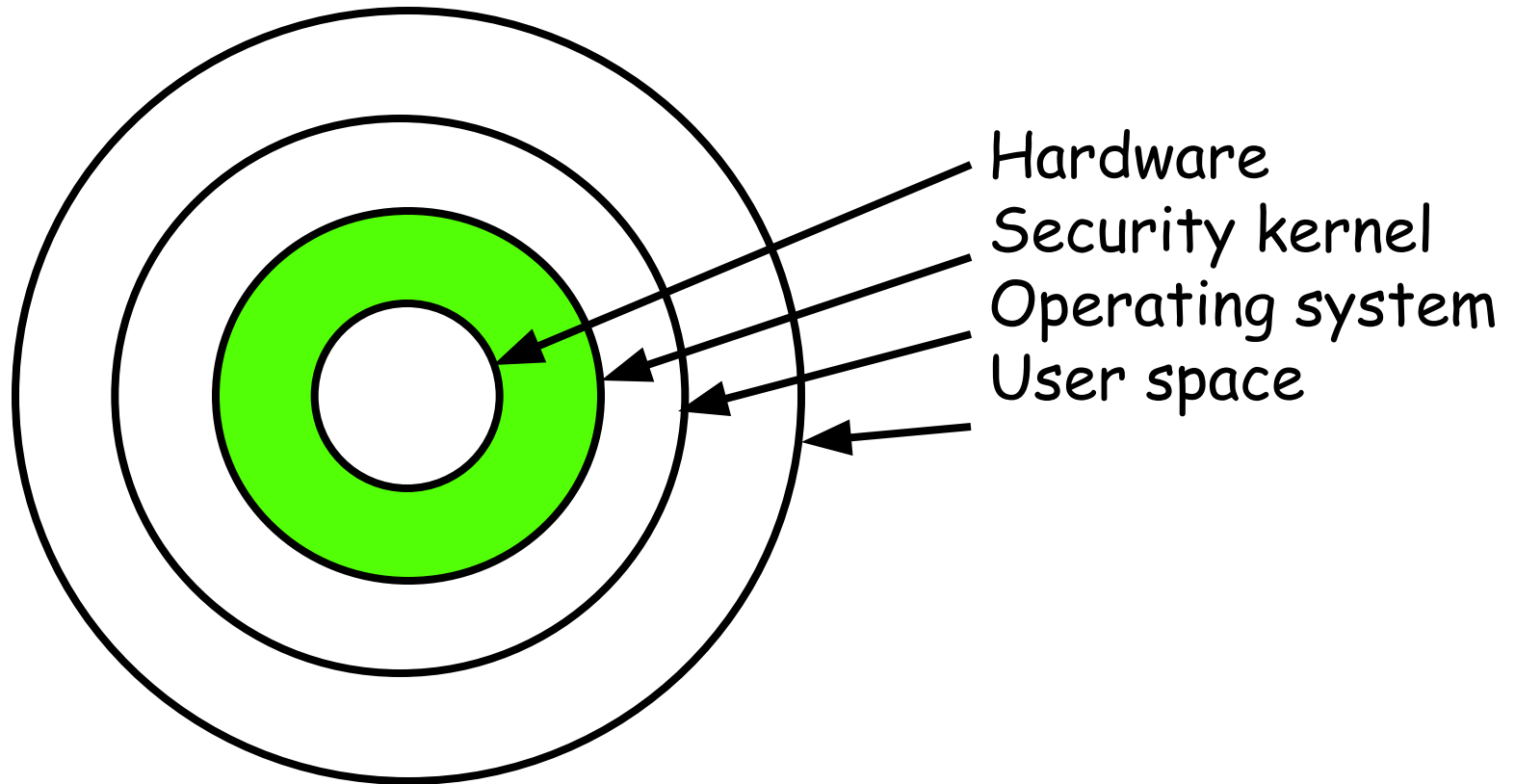
- ❑ Security may occur many places within OS
- ❑ Ideally, design security kernel first, and build the OS around it
 - Reality is usually the other way around
- ❑ Example of a trusted OS: **SCOMP**
 - Developed by Honeywell
 - Less than 10,000 LOC in SCOMP security kernel
 - Win XP has 40,000,000 lines of code!

Poor TCB Design



Problem: No clear security **layer**

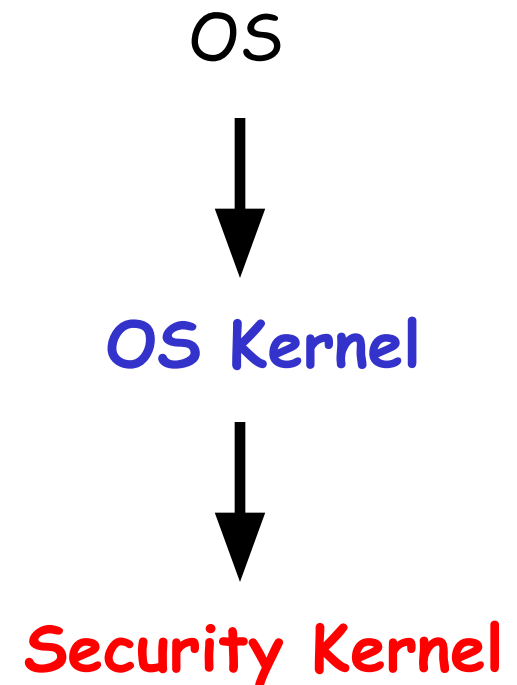
Better TCB Design



Security kernel is **the** security layer

Trusted OS Summary

- ❑ Trust implies reliance
- ❑ TCB (trusted computing base) is everything in OS we rely on for security
- ❑ If everything outside TCB is subverted, we still have trusted system
- ❑ If TCB subverted, security is broken



NGSCB

Next Generation Secure Computing Base

- ❑ **NGSCB** pronounced "n-scub" (the G is silent)
- ❑ Was supposed to be part of Vista OS
 - o Vista was once known as **Longhorn...**
- ❑ **TCG** (Trusted Computing Group)
 - o Led by Intel, TCG makes special hardware
- ❑ NGSCB is the part of Windows that will interface with TCG hardware
- ❑ TCG/NGSCB formerly TCPA/Palladium
 - o Why the name changes?



NGSCB

- ❑ The original motivation for TCPA/Palladium was digital rights management (DRM)
- ❑ Today, TCG/NGSCB is promoted as general security-enhancing technology
 - DRM just one of many potential applications
- ❑ Depending on who you ask, TCG/NGSCB is
 - Trusted computing
 - Treacherous computing

Motivation for TCG/NGSCB

- ❑ **Closed systems:** Game consoles, etc.
 - Good at protecting secrets (tamper resistant)
 - Good at forcing people to pay for software
 - Limited flexibility
- ❑ **Open systems:** PCs
 - Incredible flexibility
 - Poor at protecting secrets
 - Very poor at defending their own software
- ❑ TCG: closed system security on open platform
- ❑ “virtual set-top box inside your PC” —Rivest

TCG/NGSCB

- ❑ TCG provides tamper-resistant hardware
 - Secure place to store cryptographic key
 - Key secure from a user with admin privileges!
- ❑ TCG hardware is in addition to ordinary hardware, not in place of it
- ❑ PC has two OSs —regular OS and special **trusted** OS to deal with TCG hardware
- ❑ NGSCB is Microsoft's trusted OS

NGSCB Design Goals

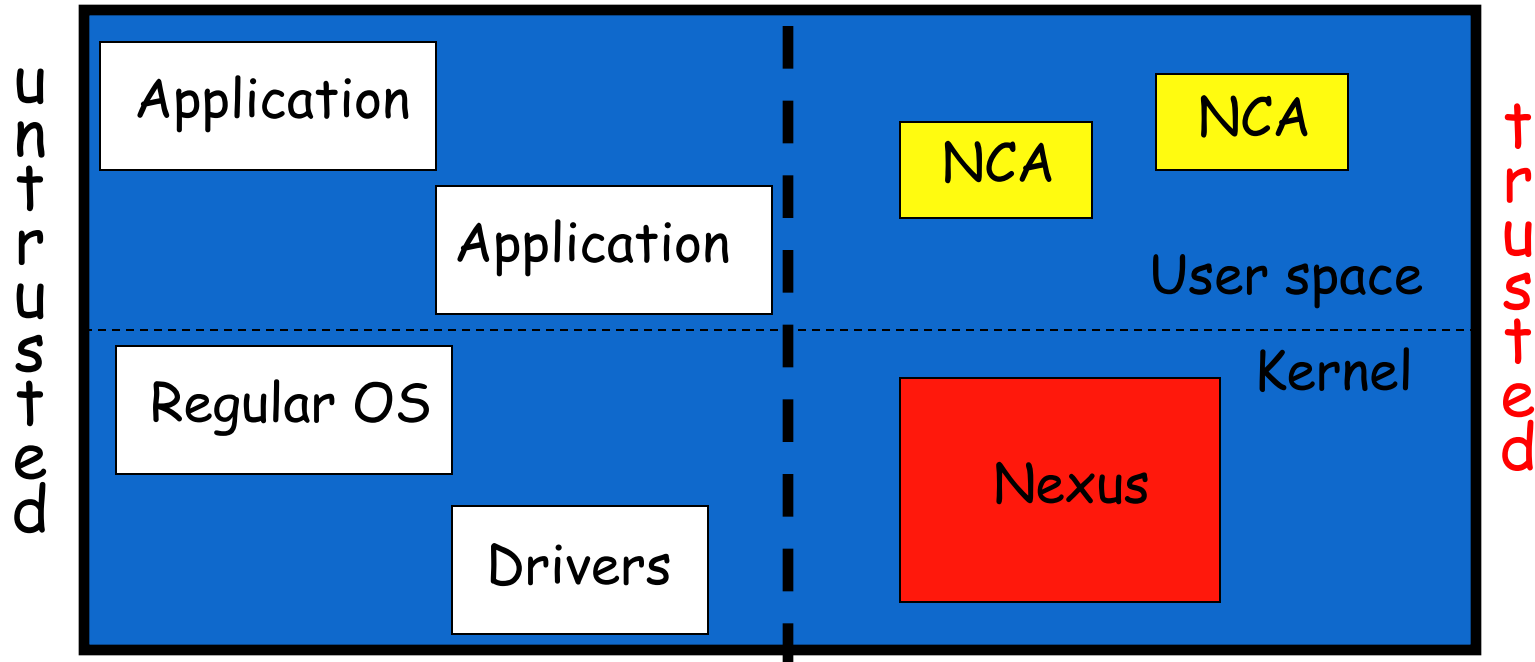
- ❑ Provide **high assurance**
 - High confidence that system behaves correctly
 - Correct behavior even if system is under attack
- ❑ Provide **authenticated operation**
 - Authenticate “things” (software, devices, etc.)
- ❑ Protection against hardware tampering is concern of TCG, not NGSCB

NGSCB Disclaimer

- ❑ Specific details are sketchy
- ❑ Based on available info, Microsoft may not have resolved all of the details
 - Maybe un-resolvable?
- ❑ What follows: author's best guesses
- ❑ This should all become much clearer in the not-too-distant future
 - At least I thought so a couple of years ago...

NGSCB Architecture

Left-hand side (LHS) Right-hand side (RHS)



- **Nexus** is the Trusted Computing Base in NGSCB
- The **NCA** (Nexus Computing Agents) talk to Nexus and LHS

NGSCB

- NGSCB has 4 “feature groups”
 1. **Strong process isolation**
 - Processes do not interfere with each other
 2. **Sealed storage**
 - Data protected (tamper resistant hardware)
 3. **Secure path**
 - Data to and from I/O protected
 4. **Attestation**
 - “Things” securely authenticated
 - Allows TCB to be extended via NCAs
- All are aimed at malicious code
- 4. also provides (secure) extensibility

NGSCB Process Isolation

- ❑ **Curtained memory**
- ❑ Process isolation and the OS
 - Protect trusted OS (Nexus) from untrusted OS
 - Isolate trusted OS from untrusted stuff
- ❑ Process isolation and NCAs
 - NCAs isolated from software they do not trust
- ❑ Trust determined by users, to an extent...
 - User **can** disable a trusted NCA
 - User **cannot** enable an untrusted NCA

NGSCB Sealed Storage

- ❑ Sealed storage contains **secret** data
 - If **code X** wants access to secret, a hash of X must be verified (integrity check of X)
 - Implemented via symmetric key cryptography
- ❑ Confidentiality of secret is protected since only accessed by trusted software
- ❑ Integrity of secret is assured since it's in sealed storage

NGSCB Secure Path

- ❑ Secure path for input
 - From keyboard to Nexus
 - From mouse to Nexus
 - From any input device to Nexus
- ❑ Secure path for output
 - From Nexus to the screen
- ❑ Uses crypto (digital signatures)

NGSCB Attestation (1)

- ❑ Secure authentication of **things**
 - Authenticate devices, services, code, etc.
 - Separate from user authentication
- ❑ Public key cryptography used
 - Certified key pair required
 - Private key not user-accessible
 - Sign and send result to remote system
- ❑ **TCB extended via attestation of NCAs**
 - This is a major feature!

NGSCB Attestation (2)

- ❑ Public key used for attestation
 - However, public key reveals the user identity
 - Using public keys, anonymity would be lost
- ❑ Trusted third party (TTP) can be used
 - TTP verifies signature
 - Then TTP vouches for signature
 - Anonymity preserved (except to TTP)
- ❑ Support for zero knowledge proofs
 - Verify knowledge of a secret without revealing it
 - Anonymity “preserved unconditionally”

NGSCB Compelling Apps (1)

- ❑ Type your Word document in Windows
 - I.e., the untrusted LHS
- ❑ Move document to trusted RHS
- ❑ Read document carefully
- ❑ Digitally sign the document
- ❑ Assured that “what you see is what you sign”
 - Practically impossible to get this on your PC

NGSCB Compelling Apps (2)

- ❑ Digital Rights Management (DRM)
- ❑ Many DRM problems solved by NGSCB
- ❑ **Protect secret** —sealed storage
 - Impossible without something like NGSCB
- ❑ **Scraping data** —secure path
 - Cannot prevent without something like NGSCB
- ❑ **Positively ID users**
 - Higher assurance with NGSCB

NGSCB According to MS

- ❑ All of Windows works on untrusted LHS
- ❑ User is in charge of...
 - Which Nexus(es) will run on system
 - Which NCAs will run on system
 - Which NCAs allowed to identify system, etc.
- ❑ No external process enables Nexus or NCA
- ❑ Nexus can't block, delete, censor data
 - NCA **does**, but NCAs **authorized** by user
- ❑ Nexus is open source

NGSCB Critics

- ❑ **Many** critics —we consider two
- ❑ Ross Anderson
 - Perhaps the most influential critic
 - Also one of the harshest critics
- ❑ Clark Thomborson
 - Lesser-known critic
 - Criticism strikes at heart of NGSCB

Anderson's NGSCB Criticism (1)

- ❑ Digital object controlled by its creator, not user of machine where it resides: Why?
 - Creator can specify the NCA
 - If user does not accept NCA, access is denied
 - Aside: This is critical for, say, MLS applications
- ❑ If Microsoft Word encrypts all documents with key only available to Microsoft products
 - Then difficult to stop using Microsoft products

Anderson's NGSCB Criticism (2)

- ❑ Files from a compromised machine could be blacklisted to, e.g., prevent music piracy
- ❑ Suppose everyone at SJSU uses same pirated copy of Microsoft Word
 - If you stop this copy from working on all NGSCB machines, SJSU users will not use NGSCB
 - Instead, make all NGSCB machines refuse to open documents created with this copy of Word...
 - ...so SJSU user can't share docs with NGSCB user...

Anderson's NGSCB Criticism (3)

- ❑ Going off the deep end...
 - "The Soviet Union tried to register and control all typewriters. NGSCB attempts to register and control all computers."
 - "In 2010 President Clinton may have two red buttons on her desk —one that sends missiles to China and another that turns off all of the PCs in China..."

Thomborson's NGSCB Criticism

- ❑ NGSCB acts like a **security guard**
- ❑ By passive observation, NGSCB "security guard" can see sensitive info
- ❑ Former student worked as security guard at apartment complex
 - By passive observations...
 - ...he learned about people who lived there

Thomborson's NGSCB Criticism

- ❑ Can NGSCB spy on you?
- ❑ According to Microsoft
 - Nexus software is public
 - NCAs can be debugged (for development)
 - NGSCB is strictly "opt in"
- ❑ Loophole?
 - Release version of NCA can't be debugged
and debug and release versions differ

NGSCB Bottom Line (1)

- ❑ NGCSB: **trusted OS** on an open platform
- ❑ Without something similar, PC may lose out
 - Particularly in entertainment-related areas
 - Copyright holders will not trust PC
 - Already lost? (iPod, Kindle, iPad, etc., etc.)
- ❑ With NGSCB, will users lose some control of their PCs?
- ❑ But NGSCB users must choose to “opt in”
 - If user does not opt in, what has been lost?

NGSCB Bottom Line (2)

- ❑ NGSCB is a **trusted system**
- ❑ **Only trusted system can break security**
 - By definition, an untrusted system is not trusted with security critical tasks
 - Also by definition, a trusted system is trusted with security critical tasks
 - If untrusted system is compromised, security is not at risk
 - If a trusted system is compromised (or simply malfunctions), security is at risk

Software Summary

- ❑ Software flaws
 - Buffer overflow
 - Race conditions
 - Incomplete mediation
- ❑ Malware
 - Viruses, worms, etc.
- ❑ Other software-based attacks

Software Summary

- ❑ Software Reverse Engineering (SRE)
- ❑ Digital Rights Management (DRM)
- ❑ Secure software development
 - Penetrate and patch
 - Open vs closed source
 - Testing

Software Summary

- ❑ Operating systems and security
 - How does OS enforce security?
- ❑ Trusted OS design principles
- ❑ Microsoft's NGSCB
 - A trusted OS for DRM

Course Summary

- ❑ Crypto
 - Symmetric key, public key, hash functions, cryptanalysis
- ❑ Access Control
 - Authentication, authorization
- ❑ Protocols
 - Simple auth., SSL, IPSec, Kerberos, GSM
- ❑ Software
 - Flaws, malware, SRE, Software development, trusted OS