

CS 458 / 658  
Computer Security and Privacy

Module 2  
Program Security

Winter 2014

# Secure programs

- Why is it so hard to write secure programs?
- A simple answer [CHE02]:
  - Axiom (Murphy):  
Programs have bugs
  - Corollary:  
Security-relevant programs have security bugs

# Module outline

- ① Flaws, faults, and failures
- ② Unintentional security flaws
- ③ Malicious code: Malware
- ④ Other malicious code
- ⑤ Nonmalicious flaws
- ⑥ Controls against security flaws in programs

# Module outline

- 1 Flaws, faults, and failures
- 2 Unintentional security flaws
- 3 Malicious code: Malware
- 4 Other malicious code
- 5 Nonmalicious flaws
- 6 Controls against security flaws in programs

# Flaws, faults, and failures

- A **flaw** is a problem with a program
- A **security flaw** is a problem that affects security in some way
  - Confidentiality, integrity, availability
- Flaws come in two types: **faults** and **failures**
- A fault is a mistake “behind the scenes”
  - An error in the code, data, specification, process, etc.
  - A fault is a **potential problem**

# Flaws, faults, and failures

- A failure is when something actually goes wrong
  - You log in to the library's web site, and it shows you someone else's account
  - "Goes wrong" means a deviation from the desired behaviour, not necessarily from the specified behaviour!
    - The specification itself may be wrong
- A fault is the programmer/specifier/inside view
- A failure is the user/outside view

# Finding and fixing faults

- How do you find a fault?
  - If a user experiences a failure, you can try to work backwards to uncover the underlying fault
  - What about faults that haven't (yet) led to failures?
  - Intentionally try to **cause** failures, then proceed as above
    - Remember to think like an attacker!
- Once you find some faults, fix them
  - Usually by making small edits (**patches**) to the program
  - This is called “penetrate and patch”
  - Microsoft's “Patch Tuesday” is a well-known example

# Problems with patching

- Patching sometimes makes things **worse!**
- Why?
  - Pressure to patch a fault is often high, causing a narrow focus on the observed failure, instead of a broad look at what may be a more serious underlying problem
  - The fault may have caused other, unnoticed failures, and a partial fix may cause inconsistencies or other problems
  - The patch for this fault may introduce new faults, here or elsewhere!
- Alternatives to patching?



# Unexpected behaviour

- When a program's behaviour is specified, the spec usually lists the things the program must do
  - The `ls` command must list the names of the files in the directory whose name is given on the command line, if the user has permissions to read that directory
- Most implementors wouldn't care if it did additional things as well
  - Sorting the list of filenames alphabetically before outputting them is fine

# Unexpected behaviour

- But from a security / privacy point of view, extra behaviours could be bad!
  - After displaying the filenames, post the list to a public web site
  - After displaying the filenames, delete the files
- When implementing a security or privacy relevant program, you should consider “and nothing else” to be implicitly added to the spec
  - “should do” vs. “shouldn’t do”
  - How would you test for “shouldn’t do”?

# Module outline

- ① Flaws, faults, and failures
- ② Unintentional security flaws
- ③ Malicious code: Malware
- ④ Other malicious code
- ⑤ Nonmalicious flaws
- ⑥ Controls against security flaws in programs

# Types of security flaws

- One way to divide up security flaws is by genesis (where they came from)
- Some flaws are **intentional**
  - **Malicious** flaws are intentionally inserted to attack systems, either in general, or certain systems in particular
    - If it's meant to attack some particular system, we call it a targeted malicious flaw
  - **Nonmalicious** (but intentional) flaws are often features that are meant to be in the system, and are correctly implemented, but nonetheless can cause a failure when used by an attacker
- Most security flaws are caused by **unintentional** program errors

# Buffer overflows

- The single most commonly exploited type of security flaw
- Simple example:

```
#define LINELEN 1024
```

```
char buffer[LINELEN];
```

```
gets(buffer);
```

or

```
strcpy(buffer, argv[1]);
```

# What's the problem?

- The gets and strcpy functions don't check that the string they're copying into the buffer **will fit in the buffer!**
- So?
- Some languages would give you some kind of exception here, and crash the program
  - Is this an OK solution?
- Not C (the most commonly used language for systems programming). C doesn't even notice something bad happened, and continues on its merry way

# Smashing The Stack For Fun And Profit

- This is a classic (read: somewhat dated) exposition of how buffer overflow attacks work.
- Upshot: if the attacker can write data past the end of an array on the stack, she can usually **overwrite things like the saved return address**. When the function returns, it will jump to any address of her choosing.
- Targets: programs on a local machine that run with setuid (superuser) privileges, or network daemons on a remote machine

# Kinds of buffer overflows

- In addition to the classic attack which overflows a buffer on the stack to jump to shellcode, there are many variants:
  - Attacks which work when a **single byte** can be written past the end of the buffer (often caused by a common off-by-one error)
  - Overflows of buffers on the heap instead of the stack
  - Jump to other parts of the program, or parts of standard libraries, instead of shellcode



# Defences against buffer overflows

- How might one protect against buffer overflows?
- Use a language with bounds checking
  - And catch those exceptions!
- Non-executable stack
  - “W $\oplus$ X” (memory page is either writable or executable, but never both)
- Stack (and sometimes code, heap, libraries) at random virtual addresses for each process
  - All mainstream OSes do this
- “Canaries” that detect if the stack has been overwritten before the return from each function
  - This is a compiler feature

# Integer overflows

- Machine integers can represent only a limited set of numbers, might not correspond to programmer's mental model
- Program assumes that integer is always positive, overflow will make (signed) integer wrap and become negative, which will violate assumption
  - Program casts large unsigned integer to signed integer
  - Result of a mathematical operation causes overflow
- Attacker can pass values to program that will trigger overflow

# Format string vulnerabilities

- Class of vulnerabilities discovered only in 2000
- Unfiltered user input is used as format string in `printf()`, `fprintf()`, `sprintf()`,...
- `printf(buffer)` instead of `printf("%s", buffer)`
  - The first one will parse `buffer` for `%`'s and use whatever is currently on the stack to process found format parameters
- `printf("%s%s%s%s")` likely crashes your program
- `printf("%x%x%x%x")` dumps parts of the stack
- `%n` will **write** to an address found on the stack
- See course readings for more

# Incomplete mediation

- Inputs to programs are often specified by untrusted users
  - Web-based applications are a common example
  - “Untrusted” to do what?
- Users sometimes mistype data in web forms
  - Phone number: 51998884567
  - Email: iang#cs.uwaterloo.ca
- The web application needs to ensure that what the user has entered constitutes a **meaningful** request
- This is called **mediation**

# Incomplete mediation

- Incomplete mediation occurs when the application accepts incorrect data from the user
- Sometimes this is hard to avoid
  - Phone number: 519-886-4567
  - This is a reasonable entry, that happens to be wrong
- We focus on catching entries that are clearly wrong
  - Not well formed
    - DOB: 1980-04-31
  - Unreasonable values
    - DOB: 1876-10-12
  - Inconsistent with other entries

# Why do we care?

- What's the security issue here?
- What happens if someone fills in:
  - DOB: 98764874236492483649247836489236492
    - Buffer overflow?
  - DOB: ' ; DROP DATABASE clients; --
    - SQL injection?
- We need to make sure that any user-supplied input falls within well-specified values, known to be safe

# SQL injection



<http://xkcd.com/327/>

# Client-side mediation

- You've probably visited web sites with forms that do **client-side** mediation
  - When you click "submit", Javascript code will first run validation checks on the data you entered
  - If you enter invalid data, a popup will prevent you from submitting it
- Related issue: client-side state
  - Many web sites rely on the client to keep state for them
  - They will put hidden fields in the form which are passed back to the server when the user submits the form



# Client-side mediation

- Problem: what if the user
  - Turns off Javascript?
  - Edits the form before submitting it? (Greasemonkey)
  - Writes a script that interacts with the web server instead of using a web browser at all?
  - Connects to the server “manually”?  
(telnet server.com 80)
- Note that the user can send arbitrary (unmediated) values to the server this way
- The user can also modify any client-side state

# Example

- At a bookstore website, the user orders a copy of the course text. The server replies with a form asking the address to ship to. This form has hidden fields storing the user's order
  - `<input type="hidden" name="isbn" value="0-13-239077-9">`  
`<input type="hidden" name="quantity" value="1">`  
`<input type="hidden" name="unitprice" value="111.00">`
- What happens if the user changes the “unitprice” value to “50.00” before submitting the form?

# Defences against incomplete mediation

- Client-side mediation is an OK method to use in order to have a friendlier user interface, but is useless for security purposes.
- You have to do **server-side mediation**, whether or not you also do client-side.
- For values entered by the user:
  - Always do very careful checks on the values of all fields
  - These values can potentially contain completely arbitrary 8-bit data (including accented chars, control chars, etc.) and be of any length
- For state stored by the client:
  - Make sure client has not modified the data in any way

# TOCTTOU errors

- TOCTTOU (“TOCK-too”) errors
  - Time-Of-Check To Time-Of-Use
  - Also known as “race condition” errors
- These errors may occur when the following happens:
  - ① User requests the system to perform an action
  - ② The system verifies the user is allowed to perform the action
  - ③ The system performs the action
- What happens if the state of the system changes between steps 2 and 3?

## Example

- A particular Unix terminal program is setuid (runs with superuser privileges) so that it can allocate terminals to users (a privileged operation)
- It supports a command to write the contents of the terminal to a log file
- It first checks if the user has permissions to write to the requested file; if so, it opens the file for writing
- The attacker makes a symbolic link:  
`logfile -> file_she_owns`
- Between the “check” and the “open”, she changes it:  
`logfile -> /etc/passwd`

# The problem

- **The state of the system changed** between the check for permission and the execution of the operation
- The file whose permissions were checked for writeability by the user (`file_she_owns`) wasn't the same file that was later written to (`/etc/passwd`)
  - Even though they had the same name (`logfile`) at different points in time
- Q: Can the attacker really “win this race”?
- A: **Yes.**

# Defences against TOCTTOU errors

- When performing a privileged action on behalf of another party, make sure all information relevant to the access control decision is **constant** between the time of the check and the time of the action (“the race”)
  - Keep a private copy of the request itself so that the request can't be altered during the race
  - Where possible, act on the object itself, and not on some level of indirection
    - e.g. Make access control decisions based on filehandles, not filenames
  - If that's not possible, use locks to ensure the object is not changed during the race

# Module outline

- 1 Flaws, faults, and failures
- 2 Unintentional security flaws
- 3 Malicious code: Malware**
- 4 Other malicious code
- 5 Nonmalicious flaws
- 6 Controls against security flaws in programs



# Malware

- Various forms of software written with malicious intent
- A common characteristic of all types of malware is that it needs to be executed in order to cause harm
- How might malware get executed?
  - User action
    - Downloading and running malicious software
    - Viewing a web page containing malicious ActiveX control
    - Opening an executable email attachment
    - Inserting a CD/DVD or USB flash drive
  - Exploiting an existing flaw in a system
    - Buffer overflows in network daemons
    - Buffer overflows in email clients or web browsers

# Viruses

- A **virus** is a particular kind of malware that infects other files
  - Traditionally, a virus could infect only executable programs
  - Nowadays, many data document formats can contain executable code (such as macros)
    - Many different types of files can be infected with viruses now
- Typically, when the file is executed (or sometimes just opened), the virus activates, and tries to infect other files with copies of itself
- In this way, the virus can spread between files, or between computers

# Infection

- What does it mean to “infect” a file?
- The virus wants to modify an existing (non-malicious) program or document (the **host**) in such a way that executing or opening it will transfer control to the virus
  - The virus can do its “dirty work” and then transfer control back to the host
- For executable programs:
  - Typically, the virus will modify other programs and copy itself to the beginning of the targets’ program code
- For documents with macros:
  - The virus will edit other documents to add itself as a macro which starts automatically when the file is opened

# Infection

- In addition to infecting other files, a virus will often try to infect the computer itself
  - This way, every time the computer is booted, the virus is automatically activated
- It might put itself in the boot sector of the hard disk
- It might add itself to the list of programs the OS runs at boot time
- It might infect one or more of the programs the OS runs at boot time
- It might try many of these strategies
  - But it's still trying to evade detection!

# Spreading

- How do viruses spread between computers?
- Usually, when the user sends infected files (hopefully not knowing they're infected!) to his friends
  - Or puts them on a p2p network
- A virus usually requires some kind of user action in order to spread to another machine
  - If it can spread on its own (via email, for example), it's more likely to be a worm than a virus

# Payload

- In addition to trying to spread, what else might a virus try to do?
- Some viruses try to evade detection by disabling any active virus scanning software
- Most viruses have some sort of **payload**
- At some point, the payload of an infected machine will activate, and something (usually bad) will happen
  - Erase (or encrypt) your hard drive
  - Subtly corrupt some of your spreadsheets
  - Install a keystroke logger to capture your online banking password
  - Start attacking a particular target website

# Spotting viruses

- When should we look for viruses?
  - As files are added to our computer
    - Via portable media
    - Via a network
  - From time to time, scan the entire state of the computer
    - To catch anything we might have missed on its way in
    - But of course, any damage the virus might have done may not be reversible
- How do we look for viruses?
  - Signature-based protection
  - Behaviour-based protection

# Signature-based protection

- Keep a list of all known viruses
- For each virus in the list, store some characteristic feature (the **signature**)
  - Most signature-based systems use features of the virus code itself
    - The infection code
    - The payload code
  - Can also try to identify other patterns characteristic of a particular virus
    - Where on the system it tries to hide itself
    - How it propagates from one place to another



# Polymorphism

- To try to evade signature-based virus scanners, some viruses are **polymorphic**
  - This means that instead of making perfect copies of itself every time it infects a new file or host, it makes a **modified** copy instead
  - This is often done by having most of the virus code encrypted
    - The virus starts with a decryption routine which decrypts the rest of the virus, which is then executed
    - When the virus spreads, it encrypts the new copy with a newly chosen random key
- How would you scan for polymorphic viruses?

# Behaviour-based protection

- Signature-based protection systems have a major limitation
  - You can only scan for viruses that are in the list!
  - But there are several brand-new viruses identified **every day**
    - One anti-virus program recognizes over *23 million* virus signatures
  - What can we do?
- Behaviour-based systems look for suspicious patterns of behaviour, rather than for specific code fragments
  - Some systems run suspicious code in a sandbox first

# False negatives and positives

- Any kind of test or scanner can have two types of errors:
  - False negatives: fail to identify a threat that is present
  - False positives: claim a threat is present when it is not
- Which is worse?
- How do you think signature-based and behaviour-based systems compare?

# Base rate fallacy

- Suppose a breathalyzer reports false drunkenness in 5% of cases, but never fails to detect true drunkenness.
- Suppose that 1 in every 1000 drivers is drunk.
- If a breathalyzer test of a random driver indicates that he or she is drunk, what is the probability that he or she really is drunk?

# Trojan horses



# Trojan horses

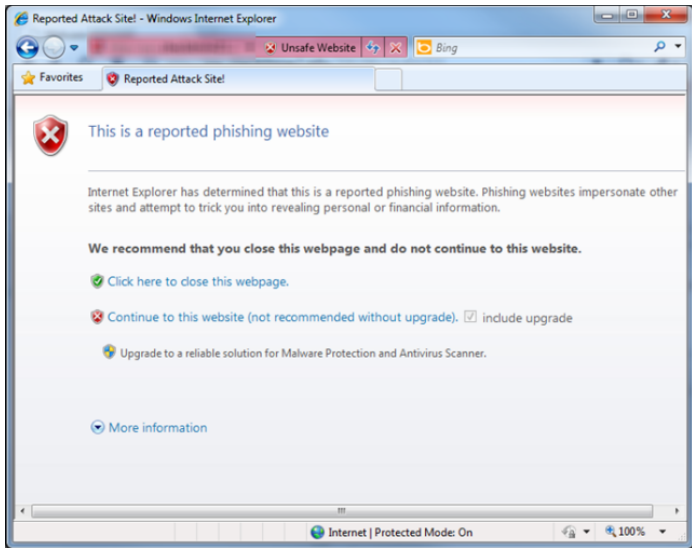
- **Trojan horses** are programs which claim to do something innocuous (and usually do), but which also hide malicious behaviour

*You're surfing the Web and you see a button on the Web site saying, "Click here to see the dancing pigs." And you click on the Web site and then this window comes up saying, "Warning: this is an untrusted Java applet. It might damage your system. Do you want to continue? Yes/No." Well, the average computer user is going to pick dancing pigs over security any day. And we can't expect them not to. — Bruce Schneier*

# Dancing pig



# Scareware



[http://static.arstechnica.com/malware\\_warning\\_2010.png](http://static.arstechnica.com/malware_warning_2010.png)



# Trojan horses

- Trojan horses:
  - Gain control by getting the user to run code of the attacker's choice, usually by also providing some code the user **wants** to run
    - "PUP" (potentially unwanted programs) are an example
    - For scareware, the user might even pay the attacker to run the code
  - The payload can be anything; sometimes the payload of a Trojan horse is itself a virus, for example
  - Trojan horses usually do not themselves spread between computers; they rely on multiple users executing the "trojaned" software
    - Better: users share the trojaned software on p2p networks

# Ransomware

CryptoLocker

## Your personal files are encrypted!

Your important files **encryption** produced on this computer: photos, videos, documents, etc. [Here](#) is a complete list of encrypted files, and you can personally verify this.

Encryption was produced using a **unique** public key **RSA-2048** generated for this computer. To decrypt files you need to obtain the **private key**.

The **single copy** of the private key, which will allow you to decrypt the files, located on a secret server on the Internet; the server will **destroy** the key after a time specified in this window. After that, **nobody and never will be able** to restore files...

**To obtain** the private key for this computer, which will automatically decrypt files, you need to pay **300 USD / 300 EUR / similar amount** in another currency.

Click «Next» to select the method of payment and the currency.

**Any attempt to remove or damage this software will lead to the immediate destruction of the private key by server.**

Private key will be destroyed on  
9/13/2013  
5:27 AM

Time left  
**71 : 19 : 53**

Next >>

<http://www.neowin.net/forum/topic/>

# Logic bombs

- A **logic bomb** is malicious code hiding in the software **already on your computer**, waiting for a certain trigger to “go off” (execute its payload)
- Logic bombs are usually written by “insiders”, and are meant to be triggered sometime in the future
  - After the insider leaves the company
- The payload of a logic bomb is usually pretty dire
  - Erase your data
  - Corrupt your data
  - Encrypt your data, and ask you to send money to some offshore bank account in order to get the decryption key!

# Logic bombs

- What is the trigger?
- Usually something the insider can affect once he is no longer an insider
  - Trigger when this particular account gets three deposits of equal value in one day
  - Trigger when a special sequence of numbers is entered on the keypad of an ATM
  - Just trigger at a certain time in the future (called a “time bomb”)

# Spotting Trojan horses and logic bombs

- Spotting Trojan horses and logic bombs is extremely tricky. Why?
- The user is **intentionally** running the code!
  - Trojan horses: the user clicked “yes, I want to see the dancing pigs”
  - Logic bombs: the code is just (a hidden) part of the software already installed on the computer
- Don't run code from untrusted sources?
- Better: prevent the payload from doing bad things
  - More on this later

# Worms

- A **worm** is a self-contained piece of code that can replicate with little or no user involvement
- Worms often use security flaws in widely deployed software as a path to infection
- Typically:
  - A worm exploits a security flaw in some software on your computer, infecting it
  - The worm immediately starts searching for other computers (on your local network, or on the Internet generally) to infect
  - There may or may not be a payload that activates at a certain time, or by another trigger

# The Morris worm

- The first Internet worm, launched by a graduate student at Cornell in 1988
- Once infected, a machine would try to infect other machines in three ways:
  - Exploit a buffer overflow in the “finger” daemon
  - Use a back door left in the “sendmail” mail daemon
  - Try a “dictionary attack” against local users’ passwords. If successful, log in as them, and spread to other machines they can access without requiring a password
- All three of these attacks were well known!
- First example of buffer overflow exploit in the wild
- Thousands of systems were offline for several days

# The Code Red worm

- Launched in 2001
- Exploited a buffer overflow in Microsoft's IIS web server (for which a patch had been available for a month)
- An infected machine would:
  - Deface its home page
  - Launch attacks on other web servers (IIS or not)
  - Launch a denial-of-service attack on a handful of web sites, including [www.whitehouse.gov](http://www.whitehouse.gov)
  - Installed a back door and a Trojan horse to try to prevent disinfection
- Infected 250,000 systems in nine hours



# The Slammer worm

- Launched in 2003
- First example of a “Warhol worm”
  - A worm which can infect nearly all vulnerable machines in just 15 minutes
- Exploited a buffer overflow in Microsoft’s SQL Server (also having a patch available)
- A vulnerable machine could be infected with a single UDP packet!
  - This enabled the worm to spread extremely quickly
  - Exponential growth, doubling every **8.5 seconds**
  - 90% of vulnerable hosts infected in 10 minutes

# Module outline

- ① Flaws, faults, and failures
- ② Unintentional security flaws
- ③ Malicious code: Malware
- ④ Other malicious code
- ⑤ Nonmalicious flaws
- ⑥ Controls against security flaws in programs

# Web bugs

- A **web bug** is an object (usually a 1x1 pixel transparent image) embedded in a web page, which is fetched from a different server from the one that served the web page itself.
- Information about you can be sent to third parties (often advertisers) without your knowledge or consent
  - IP address
  - Contents of cookies (to link cookies across web sites)
  - Any personal info the site has about you

# Web bug example

- On the quicken.com home page:
  - `<IMG WIDTH="1" HEIGHT="1" src="http://app.insightgrit.com/1/nat?id=79152388778&ref=http://www.eff.org/Privacy/Marketing/web_bug.html&z=668951&purl=http://quicken.intuit.com/">`
- What information can you see being sent to insightgrit.com?

# “Malicious code” ?

- Why do we consider web bugs “malicious code” ?
- This is an issue of privacy more than of security
- The web bug instructs your browser to behave in a way contrary to the principle of informational self-determination
  - Much in the same way that a buffer overflow attack would instruct your browser to behave in a way contrary to the security policy

# Leakage of your identity

- With the help of cookies, an advertiser can learn what websites a person is interested in
- But the advertiser cannot learn person's identity
- ... unless the advertiser can place ads on a social networking site
- Content of HTTP request for Facebook ad:  
GET [pathname of ad]  
Host: ad.doubleclick.net  
Referer: http://www.facebook.com/  
profile.php?id=123456789&ref=name  
Cookie: id=2015bdfb9ec...

# Back doors

- A **back door** (also called a **trapdoor**) is a set of instructions designed to bypass the normal authentication mechanism and allow access to the system to anyone who knows the back door exists
  - Sometimes these are useful for debugging the system, but **don't forget to take them out before you ship!**
- Fanciful examples:
  - "Reflections on Trusting Trust" (mandatory reading)
  - "The Net"
  - "WarGames"

# Examples of back doors

- Real examples:
  - Debugging back door left in sendmail
  - Back door planted by Code Red worm
  - Port knocking
    - The system listens for connection attempts to a certain pattern of (closed) ports. All those connection attempts will fail, but if the right pattern is there, the system will open, for example, a port with a root shell attached to it.
  - Attempted hack to Linux kernel source code
    - ```
if ((options == (_WCLONE|_WALL)) &&  
    (current->uid = 0))  
    retval = -EINVAL;
```



# Sources of back doors

- Forget to remove them
- Intentionally leave them in for testing purposes
- Intentionally leave them in for maintenance purposes
  - Field service technicians
- Intentionally leave them in for malicious purposes
  - Note that malicious users can use back doors left in for non-malicious purposes, too!

# Salami attacks

- A **salami attack** is an attack that is made up of many smaller, often considered inconsequential, attacks
- Classic example: send the fractions of cents of round-off error from many accounts to a single account owned by the attacker
- More commonly:
  - Credit card thieves make very small charges to very many cards
  - Clerks slightly overcharge customers for merchandise
  - Gas pumps misreport the amount of gas dispensed

# Privilege escalation

- Most systems have the concept of differing levels of privilege for different users
  - Web sites: everyone can read, only a few can edit
  - Unix: you can write to files in your home directory, but not in `/usr/bin`
  - Mailing list software: only the list owner can perform certain tasks
- A **privilege escalation** is an attack which raises the privilege level of the attacker (beyond that to which he would ordinarily be entitled)

# Sources of privilege escalation

- A privilege escalation flaw often occurs when a part of the system that **legitimately** runs with higher privilege can be tricked into executing commands (with that higher privilege) on behalf of the attacker
  - Buffer overflows in setuid programs or network daemons
  - Component substitution (see attack on search path in textbook)
- Also: the attacker might trick the system into thinking he is in fact a legitimate higher-privileged user
  - Problems with authentication systems
    - “-froot” attack

# Rootkits

- A **rootkit** is a tool often used by “script kiddies”
- It has two main parts:
  - A method for gaining unauthorized root / administrator privileges on a machine (either starting with a local unprivileged account, or possibly remotely)
    - This method usually exploits some known flaw in the system that the owner has failed to correct
    - It often leaves behind a back door so that the attacker can get back in later, even if the flaw is corrected
  - A way to hide its own existence
    - “Stealth” capabilities
    - Sometimes just this stealth part is called the rootkit

# Stealth capabilities

- How do rootkits hide their existence?
  - Clean up any log messages that might have been created by the exploit
  - Modify commands like `ls` and `ps` so that they don't report files and processes belonging to the rootkit
  - Alternately, modify the **kernel** so that no user program will ever learn about those files and processes!

## Example: Sony XCP

- Mark Russinovich was developing a rootkit scanner for Windows
- When he was testing it, he discovered his machine already had a rootkit on it!
- The source of the rootkit turned out to be Sony audio CDs equipped with XCP “copy protection”
- When you insert such an audio CD into your computer, it contains an `autorun.exe` file which automatically executes
- `autorun.exe` installs the rootkit

## Example: Sony XCP

- The “primary” purpose of the rootkit was to modify the CD driver in Windows so that any process that tried to read the contents of an XCP-protected CD into memory would get garbled output
- The “secondary” purpose was to make itself hard to find and uninstall
  - Hid all files and processes whose names started with \$sys\$
- After people complained, Sony eventually released an uninstaller
  - But running the uninstaller left a back door on your system!



# Keystroke logging

- Almost all of the information flow from you (the user) to your computer (or beyond, to the Internet) is via the keyboard
  - A little bit from the mouse, a bit from devices like USB keys
- An attacker might install a **keyboard logger** on your computer to keep a record of:
  - All email / IM you send
  - All passwords you type
- This data can then be accessed locally, or it might be sent to a remote machine over the Internet

# Who installs keyboard loggers?

- Some keyboard loggers are installed by malware
  - Capture passwords, especially banking passwords
  - Send the information to the remote attacker
- Others are installed by one family member to spy on another
  - Spying on children
  - Spying on spouses
  - Spying on boy/girlfriends

# Kinds of keyboard loggers

- Application-specific loggers:
  - Record only those keystrokes associated with a particular application, such as an IM client
- System keyboard loggers:
  - Record all keystrokes that are pressed (maybe only for one particular target user)
- Hardware keyboard loggers:
  - A small piece of hardware that sits between the keyboard and the computer
    - Works with any OS
    - Completely undetectable in software

# Interface illusions

- You use user interfaces to control your computer all the time
- For example, you drag on a scroll bar to see offscreen portions of a document
- But what if that scrollbar isn't really a scrollbar?
- What if dragging on that “scrollbar” really dragged a program (from a malicious website) into your “Startup” folder (in addition to scrolling the document)?
  - This really happened

# Interface Illusion by Conficker worm



# Interface illusions

- We expect our computer to behave in certain ways when we interact with “standard” user interface elements.
- But often, malicious code can make “nonstandard” user interface elements in order to trick us!
- We think we’re doing one thing, but we’re really doing another
- How might you defend against this?

# Phishing

- **Phishing** is an example of an interface illusion
- It looks like you're visiting Paypal's website, but you're really not.
  - If you type in your password, you've just given it to an attacker
- Advanced phishers can make websites that look every bit like the real thing
  - Even if you carefully check the address bar, or even the SSL certificate!

# Man-in-the-middle attacks

- Keyboard logging, interface illusions, and phishing are examples of **man-in-the-middle attacks**
- The website/program/system you're communicating with isn't the one you **think** you're communicating with
- A man-in-the-middle intercepts the communication from the user, and then passes it on to the intended other party
  - That way, the user thinks nothing's wrong, because his password works, he sees his account balances, etc.



# Man-in-the-middle attacks

- But not only is the man-in-the-middle able to see (and record) everything you're doing, and can capture passwords, but once you've authenticated to your bank (for example), the man-in-the-middle can **hijack** your session to insert malicious commands
  - Make a \$700 payment to attacker@evil.com
- You won't even see it happen on your screen, and if the man-in-the-middle is clever enough, he can edit the results (bank balances, etc.) being displayed to you so that there's no visible record (to you) that the transaction occurred
  - Stealthy, like a rootkit

# Module outline

- ① Flaws, faults, and failures
- ② Unintentional security flaws
- ③ Malicious code: Malware
- ④ Other malicious code
- ⑤ Nonmalicious flaws**
- ⑥ Controls against security flaws in programs

# Covert channels

- Suppose Alice has access to very sensitive information, and Eve is an attacker who wants it
  - Medical information
  - Banking information
  - Alice's own password
- Assume that Eve can even arrange for malicious code to be running on Alice's machine
  - Trojan horse, logic bomb, etc.

# Covert channels

- Normally, Eve would just have the Trojan horse send the sensitive data to her over the Internet
- But Alice is too clever for that!
  - She closely watches all Internet traffic from her computer
  - Better, she doesn't connect her computer to the Internet at all!
- How does Eve get Alice's data?

# Covert channels

- If there's no information at all that goes from Alice to somewhere Eve can get it, there's really nothing Eve can do.
  - But this is rare
- Suppose Alice publishes a weekly report summarizing some (nonsensitive) statistics
- Eve can “hide” the sensitive data in that report!
  - Modifications to spacing, wording, or the statistics itself
  - This is called a **covert channel**
  - See pages 150–155 of the text for an example

# Side channels

- What if Eve can't get Trojaned software on Alice's computer in the first place?
- It turns out there are some very powerful attacks called **side channel** attacks
  - Eve watches how Alice's computer behaves when processing the sensitive data
  - Eve usually has to be somewhere in the physical vicinity of Alice's computer to pull this off
    - But not always!

# Side channels

- Eve can learn information about what Alice's computer is doing (and what data it is processing) by looking at:
  - RF emissions
  - Power consumption
  - Audio emissions
  - Reflected light from a CRT
  - Reflection of screen in Alice's eyeball
  - Time it takes for Alice's computer to perform a computation
  - Shared CPU cache
- These are especially powerful attacks when "Alice's computer" is a smart card (like a SIM chip or satellite TV card) that stores some kind of secret but is physically in Eve's possession

# Module outline

- ① Flaws, faults, and failures
- ② Unintentional security flaws
- ③ Malicious code: Malware
- ④ Other malicious code
- ⑤ Nonmalicious flaws
- ⑥ Controls against security flaws in programs



# The picture so far

- We've looked at a large number of ways an attacker can compromise program security
  - Exploit unintentional flaws
  - Exploit intentional, but nonmalicious, behaviour of the system
  - Introduce malicious code, including malware
- The picture looks pretty bleak
- Our job is to control these threats
  - It's a tough job

# Software lifecycle

- Software goes through several stages in its lifecycle:
  - Specification
  - Design
  - Implementation
  - Change management
  - Code review
  - Testing
  - Documentation
  - Maintenance
  
- At which stage should security controls be considered?

# Security controls—Design

- How can we design programs so that they're less likely to have security flaws?
- Modularity
- Encapsulation
- Information hiding
- Mutual suspicion
- Confinement

# Modularity

- Break the problem into a number of small pieces (“modules”), each responsible for a single subtask
- The complexity of each piece will be smaller, so each piece will be far easier to check for flaws, test, maintain, reuse, etc.
- Modules should have low **coupling**
  - A coupling is any time one module interacts with another module
  - High coupling is a common cause of unexpected behaviours in a program

# Encapsulation

- Have the modules be mostly self-contained, sharing information only as necessary
- This helps reduce coupling
- The developer of one module should not need to know how a different module is implemented
  - She should only need to know about the published interfaces to the other module (the API)

# Information hiding

- The internals of one module should not be visible to other modules
- This is a stronger statement than encapsulation: the implementation and internal state of one module should be **hidden** from developers of other modules
- This prevents accidental reliance on behaviours not promised in the API
- It also hinders some kinds of malicious actions by the developers themselves!

# Mutual suspicion

- It's a good idea for modules to check that their inputs are sensible before acting on them
- Especially if those inputs are received from untrusted sources
  - Where have we seen this idea before?
- But also as a defence against flaws in, or malicious behaviour on the part of, other modules
  - Corrupt data in one module should be prevented from corrupting other modules

# Confinement

- Similarly, if Module A needs to call a potentially untrustworthy Module B, it can **confine** it (also known as **sandboxing**)
  - Module B is run in a limited environment that only has access to the resources it absolutely needs
- This is especially useful if Module B is code downloaded from the Internet
- Suppose all untrusted code were run in this way
  - What would be the effect?



# Security controls—Implementation

- When you're actually coding, what can you do to control security flaws?
- Don't use C (but this might not be an option)
- Static code analysis
- Formal methods
- Genetic diversity
- Finally: learn more!

# Static code analysis

- There are a number of software products available that will help you find security flaws in your code
  - These work for various languages, including C, C++, Java, Perl, PHP, Python
- They often look for things like buffer overflows, but some can also point out TOCTTOU and other flaws
- These tools are not perfect!
  - They're mostly meant to find suspicious things for you to look at more carefully
  - They also miss things, so they can't be your only line of defence

# Formal methods

- Instead of looking for suspicious code patterns, **formal methods** try to **prove** that the code does exactly what it's supposed to do
  - And you thought the proofs in your math classes were hard?
  - Unfortunately, we can show that this is impossible to do in general
    - But that doesn't mean we can't find large classes of useful programs where we can do these proofs in particular
- Usually, the programmer will have to “mark up” her code with assertions or other hints to the theorem proving program
  - This is time-consuming, but if you get a proof out, you can really believe it!

# Genetic diversity

- The reason worms and viruses are able to propagate so quickly is that there are many, many machines running **the same vulnerable code**
  - The malware exploits this code
- If there are lots of different HTTP servers, for example, there's unlikely to be a common flaw
- This is the same problem as in agriculture
  - If everyone grows the same crop, they can all be wiped out by a single virus

# Learn more about software security

- We barely scratched the surface in this course
- If you are thinking about becoming a software developer, get one of these books:
- “Building Secure Software - How to Avoid Security Problems the Right Way” by John Viega and Gary McGraw
- “Writing Secure Code (Second Edition)” by Michael Howard and David LeBlanc

# Security controls—Change management

- Large software projects can have dozens or hundreds of people working on the code
- Even if the code's secure today, it may not be tomorrow!
- If a security flaw does leak into the code, where did it come from?
  - Not so much to assign blame as to figure out how the problem happened, and how to prevent it from happening again

# Source code and configuration control

- Track all changes to either the source code or the configuration information (what features to enable, what version to build, etc.) in some kind of management system
- There are dozens of these; you've probably used at least a simple one before
  - CVS, Subversion, git, darcs, Perforce, Mercurial, Bitkeeper, ...
- Remember that attempted backdoor in the Linux source we talked about last time?
  - Bitkeeper noticed a change to the source repository that didn't match any valid checkin

## Security controls—Code review

- Empirically, code review is the single most effective way to find faults once the code has been written
- The general idea is to have people other than the code author look at the code to try to find any flaws
- This is one of the benefits often touted for open-source software: anyone who wants to can look at the code
  - But this doesn't mean people actually do!
  - Even open-source security vulnerabilities can sit undiscovered for years, in some cases



# Kinds of code review

- There are a number of different ways code review can be done
- The most common way is for the reviewers to just be given the code
  - They look it over, and try to spot problems that the author missed
  - This is the open-source model

# Guided code reviews

- More useful is a guided walk-through
  - The author explains the code to the reviewers
  - Justifies why it was done this way instead of that way
  - This is especially useful for **changes** to code
    - Why each change was made
    - What effects it might have on other parts of the system
    - What testing needs to be done
- Important for safety-critical systems!

# “Easter egg” code reviews

- One problem with code reviews (especially unguided ones) is that the reviewers may start to believe there's nothing there to be found
  - After pages and pages of reading without finding flaws (or after some number have been found and corrected), you really just want to say it's fine
- A clever variant: the author **inserts intentional flaws** into the code
  - The reviewers now know there **are** flaws
  - The theory is that they'll look harder, and are more likely to find the **unintentional** flaws
  - It also makes it a bit of a game

# Security controls—Testing

- The goal of testing is to make sure the implementation meets the specification
- But remember that in security, the specification includes “and nothing else”
  - How do you test for that?!
- Two main strategies:
  - Try to make the program do unspecified things just by doing unusual (or attacker-like) things to it
  - Try to make the program do unspecified things by taking into account the design and the implementation

# Black-box testing

- A test where you just have access to a completed object is a **black-box** test
  - This object might be a single function, a module, a program, or a complete system, depending on at what stage the testing is being done
- What kinds of things can you do to such an object to try to get it to misbehave?
- `int sum(int inputs[], int length)`

# Fuzz testing

- One easy thing you can do in a black-box test is called **fuzz testing**
- Supply completely random data to the object
  - As input in an API
  - As a data file
  - As data received from the network
  - As UI events
- This causes programs to crash surprisingly often!
  - These crashes are violations of Availability, but are often indications of an even more serious vulnerability

# White-box testing

- If you're testing conformance to a specification by taking into account knowledge of the design and implementation, that's **white-box** testing
  - Also called **clear-box** testing
- Often tied in with code review, of course
- White-box testing is useful for **regression testing**
  - Make a comprehensive set of tests, and ensure the program passes them
  - When the next version of the program is being tested, run all these tests again

# Security controls—Documentation

- How can we control security vulnerabilities through the use of documentation?
- Write down the choices you made
  - And why you made them
- Just as importantly, write down things you tried that **didn't work!**
  - Let future developers learn from your mistakes
- Make checklists of things to be careful of
  - Especially subtle and non-obvious security-related interactions of different components



# Security controls—Maintenance

- By the time the program is out in the field, one hopes that there are no more security flaws
  - But there probably are
- We've talked about ways to control flaws when modifying programs
  - Change management, code review, testing, documentation
- Is there something we can use to try to limit the number of flaws that make it out to the shipped product in the first place?

# Standards, process, and audit

- Within an organization, have rules about how things are done at each stage of the software lifecycle
- These rules should incorporate the controls we've talked about earlier
- These are the organization's **standards**
- For example:
  - What design methodologies will you use?
  - What kind of implementation diversity?
  - Which change management system?
  - What kind of code review?
  - What kind of testing?

# Standards, process, and audit

- Make formal **processes** specifying how each of these standards should be implemented
  - For example, if you want to do a guided code review, who explains the code to whom? In what kind of forum? How much detail?
- Have **audits**, where somebody (usually external to the organization) comes in and verifies that you're following your processes properly
- This doesn't guarantee flaw-free code, of course!

# Recap

- Flaws, faults, and failures
- Unintentional security flaws
- Malicious code: Malware
- Other malicious code
- Nonmalicious flaws
- Controls against security flaws in programs

# Recap

- Various controls applicable to each of the stages in the software development lifecycle
- To get the best chance of controlling all of the flaws:
  - Standards describing the controls to be used
  - Processes implementing the standards
  - Audits ensuring adherence to the processes