

- Malicious Logic
 - Viruses
- Buffer overflow

*CSCI283 Fall 2003 Lecture 5
GWU*

*Draws extensively from Memon's notes, Brooklyn Poly
And text, Chapter 3*

***YOU ARE EXPECTED TO READ CHAPTER 3 FROM
THE TEXT IN ADDITION TO THIS***

Announcements

CS COLLOQUIUM PRESENTATION!!

Time: Oct 8, Wed, 2:00PM-3:00PM

Location: Room 736 Academic Center

Refreshments!

Speaker: Dr. Steve Crocker, CEO
Shinkuro, Inc.

Title: The Shinkuro Peer-to-Peer File Sharing System:
Experiments using Simplex Channels for Replication

Program Security

- Secure Programs: behave as expected
 - Unexpected behaviour is a “program security flaw”
 - Happens because of an existing “vulnerability”
- IEEE Terminology
 - Human error →
 - Fault (incorrect code, internal, professional’s view) →
 - Failure (incorrect system behaviour; external, user’s/lay person’s view)

Patching

- One way of addressing faults: test, discover faults, patch them
- Problems:
 - No guarantee all faults are found
 - No guarantee the patch does not add another fault
 - Pressure leads to hurried patches
 - Because the entire system cannot be redesigned, there's a limit to how much a single patch can fix because it is constrained not to affect the rest of the system (for example, a definition of a variable that is passed on to several different modules, but creates a fault only in one)
 - Performance provides pragmatic limits

Faults will always exist

- Human error
- Complexity of system
- The study of security finds more possibilities for flaws while software engineering proceeds to find new software techniques – i.e. neither field is done with what they are doing, that the other can be expected to address it completely.
- Non-malicious and malicious faults

Malicious Logic

- Malicious logic: “*Hardware, software, or firmware capable of performing an unauthorized function on an information system.*”
NSTISSI 4009
- Usually violates security policy of a system
- Malicious logic is also known as malicious code or malware
- Unintentionally faulty code can cause the same/similar effects

Types of malicious logic (existing since at least 1970)

- Virus (*Vital Information Resources Under Siege*)
 - Self replicating code, parasitic (attaches to “good” code)
 - Can be
 - “resident” (attaches itself to memory and can execute after its host program is done) or
 - “transient” (active only while its host is executing)
- Trojan Horses
 - Program with overt effects and covert effects

Types of malicious logic – contd.

- Worms
 - Self replicating, spread through networks
 - Stand-alone, not attached to another piece of logic
- Logic Bombs
 - Waits for a trigger condition and “detonates”
 - Time bomb!
- Trapdoors
 - Alternative means of executing code
 - Intentional – legitimate and malicious purposes
 - Exploits – exploits of faulty code, buffer overflow, format string

Types of malicious logic – contd.

- ActiveX, Java code
 - Execution of malicious code via Java applets, ActiveX scripts
 - Malicious mobile code
- Bacteria and rabbits
 - Virus or worm that absorbs all of some class of resource
 - For example: self-replicating piece of code fills up disk
- Easter Eggs (<http://www.eeggs.com/>)
 - Can we categorize them as malicious logic?
 - Does PowerPoint have Easter eggs?
- Hybrids
 - Usually a mixture of above
 - An Easter egg + Worm = Easter worm

Roadmap

- Virus (used as a generic term for malicious code)
 - Types of viruses
 - Means of attaching
 - Anatomy of a simple virus
 - More sophisticated virus
 - Virus detection methods
 - Antivirus mechanisms
- Buffer overflow, format string vulnerabilities

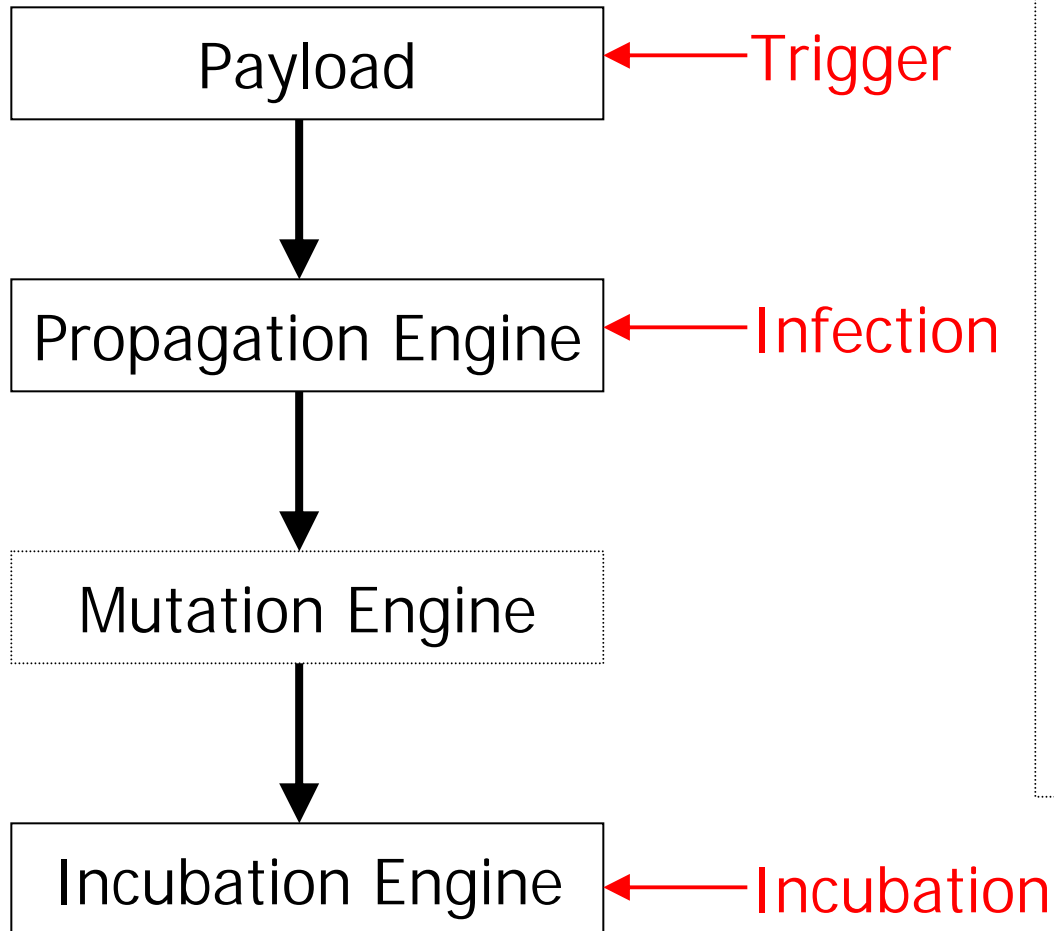
Types of virus

- Classification by where they attach
 - Boot sector viruses
 - Parasitic viruses
- Classification by type of code
 - Binary viruses: usually written in assembly language then assembled to form executable image (binary file); attaches to other binary files or boot sector.
 - Macro viruses: written in high-level macro language then interpreted (possibly after pre-processing); attaches to other files that support same macro language (e.g. Outlook, Emacs)

Types of viruses – contd.

- A general classification
 - Boot sector viruses
 - Modify and reside in boot sector
 - Parasitic viruses
 - Attach itself to files
 - Polymorphic viruses
 - Mutate like biological viruses
 - Stealth Viruses
 - Hard to detect
 - TSRs (Terminate Stay Resident)
 - Memory resident viruses
 - LKMs (Loadable Kernel Modules)
 - Future of Unix based viruses
 - Multi-partite
 - Hybrids of above

Virus logic



Infection:

Infection is the act of replicating from a host to another host

Incubation:

The time between infection and activation of payload

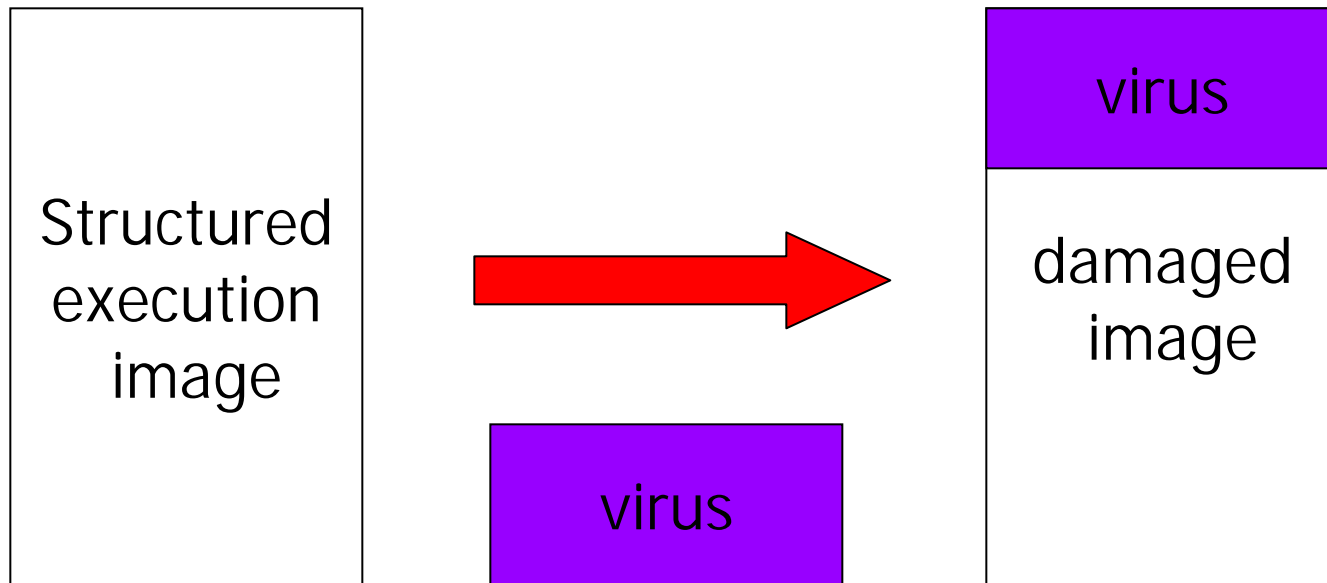
Virulence:

Number of infections per copy

Virus logic – contd.

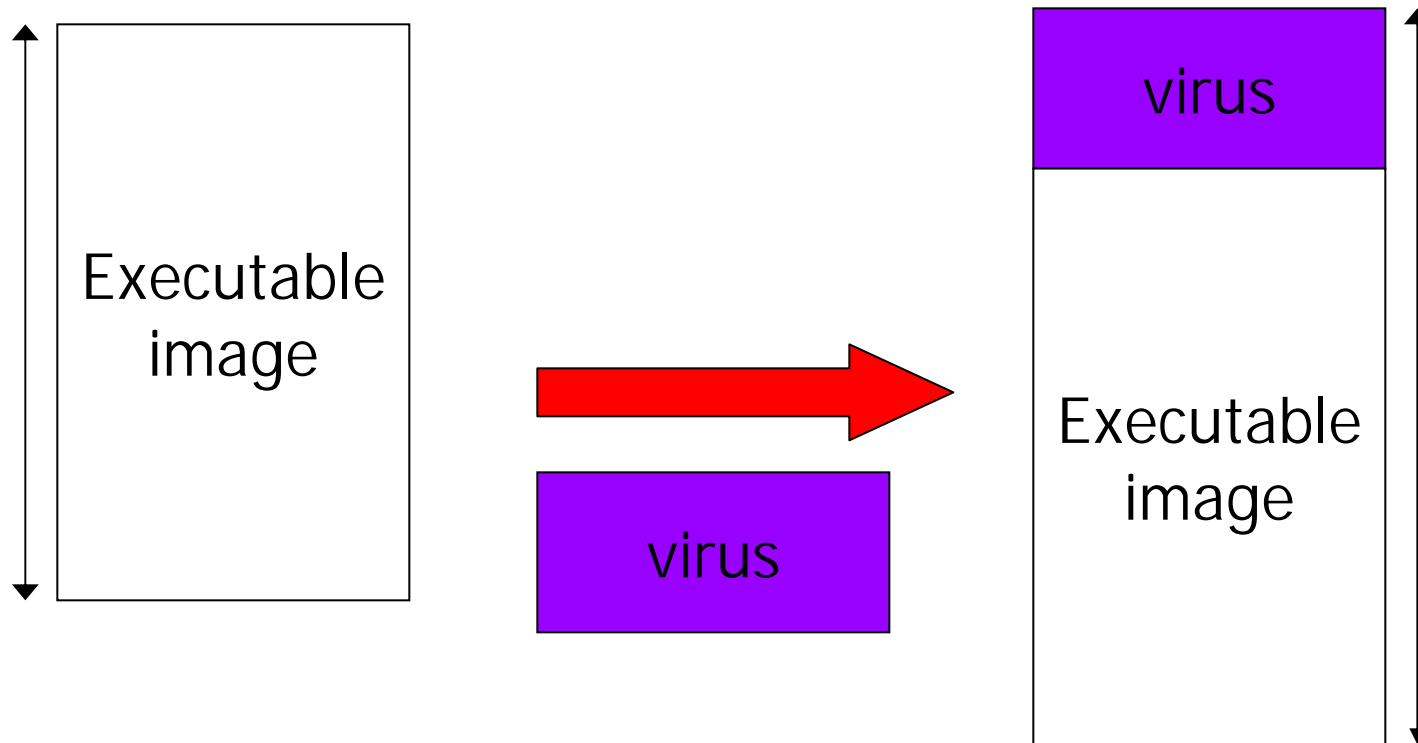
- Virus includes code to
 - Search for files to infect
 - Replicate
 - Make copy of self
 - Attach to file/boot sector
 - Payload (check trigger and do badness)
 - Measures to allude detection
 - Ideally, should execute quickly then pass control to infected program's normal code
 - Intercept system calls
 - Fool antiviral tools

Means of attaching: overwriting (virus *replaces* part of program)



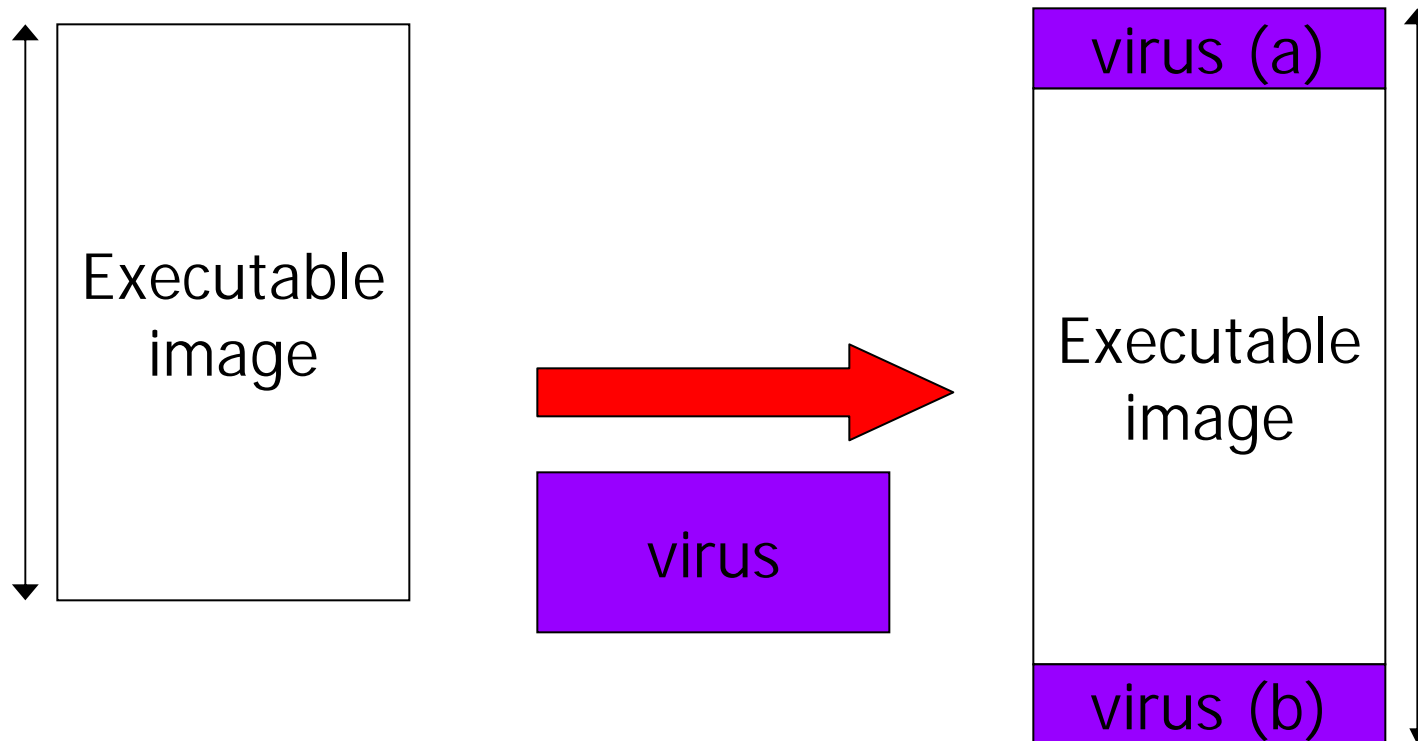
- Virus overwrites an executable file
- Easiest mechanism
- Since original program is damaged easily detected

Means of attaching: at the beginning (virus is *appended* to program)



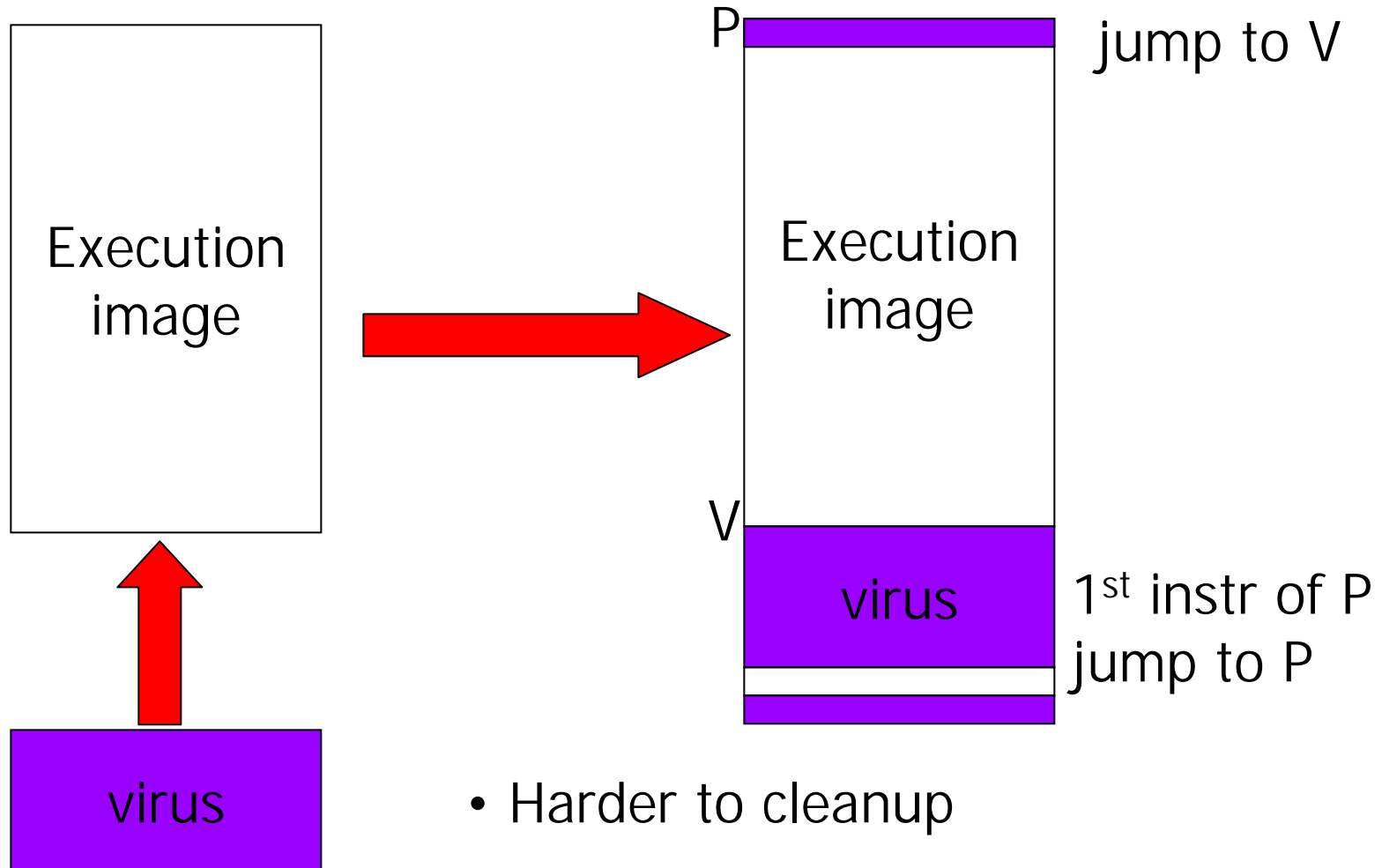
- Improved stealth because original program is intact
- If original program is large, copying it may be slow
- File size grows if multiple infections occur

Means of attaching: beginning and end (virus *surrounds* program)

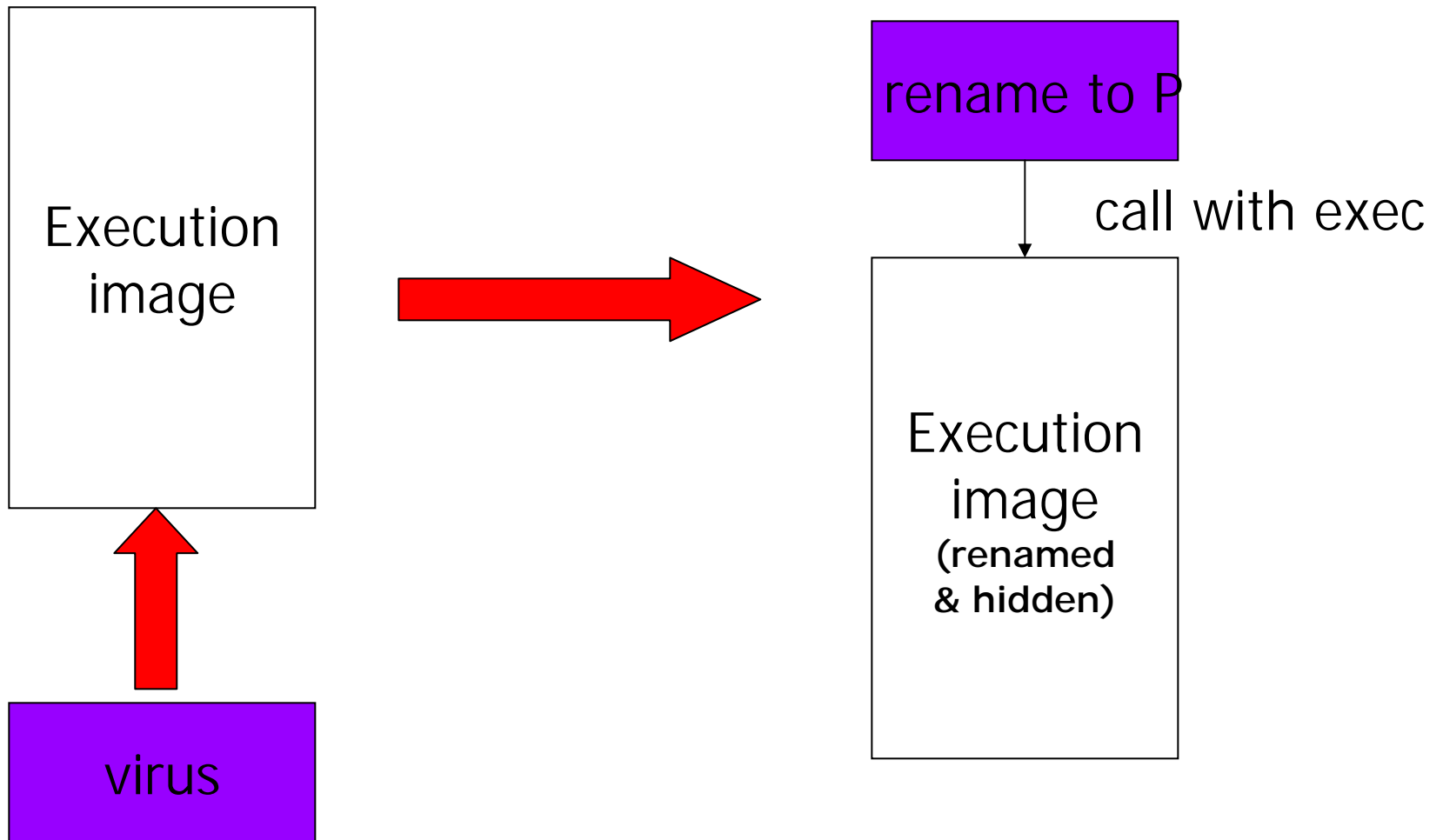


- Properties of appended virus +
- Ability to clean up and avoid detection
- Example: attach to program that constructs file lists with sizes; modify after program has ended

Means of attaching: intersperse (virus is *integrated* into program)



Means of attaching: companions



Invoking a virus

- Virus invoked because:
 - It has replaced part of a program code within the file structure
 - It has appended itself to the code within a file
 - It has overwritten the file in storage
 - It has changed the pointer in the file table, so that it is located instead of a particular file
 - It has changed the table of pointers to typical operating system parts (such as interrupt handler)

Boot sector virus

- Computer starts with firmware testing all hardware and then initializing a specified OS and transferring control to it.
- Code copies the OS from disk to memory; starts with bootstrap loader, which is a small set of instructions that then copies the rest of the OS. Initial part of bootstrap loader is contained in boot sector
- Because OS length is not pre-determined, and to allow flexibility, the bootstrap loader consists of non-contiguous blocks on disk chained together with pointers.
- Virus can easily insert itself in the chain, on disk.
- Very effective, as difficult to detect (OS files hidden, virus detection not yet activated)
- Trusted boot

Memory residents or TSRs (*Terminate and Stay Resident*)

- Infect memory-resident code (e.g. frequently used parts of the OS), which remains in memory while the computer is running
- Resident code usually activated many times, giving virus many opportunities to spread
- Example: attach to interrupt-handler and check whether any new floppies have been inserted; if so, infect floppy
- Also many other homes for viruses: libraries, application program startup macros, compilers, virus detection software!

Five major detection methods

- Integrity checking
 - Look for modified files by comparing old and new checksum
 - No software updates required
 - Requires maintenance of virus free checksums
 - Unable to detect passive, active stealth viruses
 - Cannot identify viruses by type or name
- Interrupt monitoring
 - Attempts to locate and prevent a viruses' interrupt calls
 - Poor system utilization
 - Obstructive, because of false positives
- Memory detection
 - Depends on recognition of known viruses' location and code in memory

Five major detection methods

- Signature scanning
 - Recognizes viruses' unique "signature": a pre-identified hex
 - Functional
 - Need to maintain current signature files and scanning engine refinements
 - False positives
- Heuristics/Rule based
 - Faster than traditional scanners
 - Uses a set of rules to effectively parse through files and identify code
 - Uses expert systems or neural networks
 - Depends on current rule-set

(Detection can be performed on-access or on-demand)

Properties of a good signature

- Should always appear in the virus, so there won't be any false negatives
- Should not appear in (m)any other files, so there won't be (m)any false positives
- Should be reasonably short, for efficient scanning
- For simple viruses like Mini-44, it's easy to find good signatures. Virus writers have responded with...

Polymorphic Viruses

- Polymorphic = “many forms”
- Goal: Foil virus scanners by changing virus code each time virus replicates, so that it will be difficult to find a good signature
- Approaches:
 - Encrypt virus with random key
 - Note: Goals and techniques are different than in the encryption techniques we studied earlier. XOR with stored key is sufficient.
 - “Mutate” virus by making small changes that don’t affect the semantics of the code (like birthday attack)
 - Nearly 2 billion guises can be evolved from a single code
 - Requires algorithm based matching instead of simple string based matching
 - Given two code segments, evaluating their semantic equivalency is an undecidable problem!

Replication of encrypted virus

- Copy decryption engine to infected file (as is)
- Select new key and copy it to the infected file
- For each byte of the encrypted portion of the virus:
 - take decrypted byte
 - encrypt it with the new key
 - copy it to the infected file
- Result: different replicas of virus have different byte patterns, so difficult to find signature

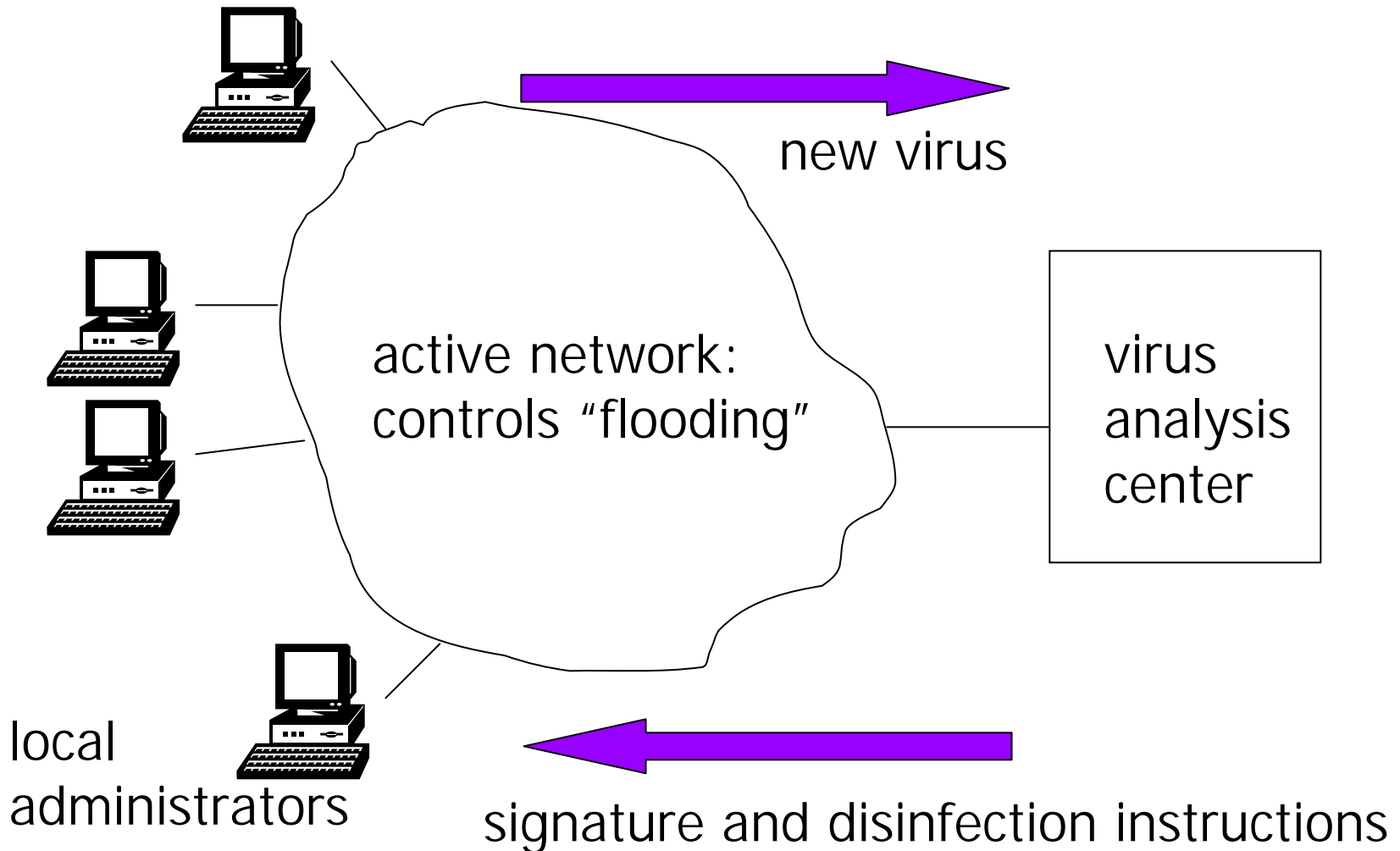
Anti-virus tools' answer to encryption

- Select the signature from the unencrypted portion of the code, I.e. the decryption engine
- Problems:
 - Anti-virus tools usually want to determine which virus is present, not just determine that some virus is present (in order to “disinfect”).
 - Can emulate the decryption then further analyze the decrypted code.
 - virus writers have responded by obscuring the encryption engine through mutations
- It's a game of cat and mouse

Virus Analysis

- Analysis of virus by human expert
 - slow: by the time signature has been extracted, posted to AV tool database, downloaded to users, virus may have spread widely.
 - pre-1995: 6 months to a year for virus to spread world-wide
 - mid-90's: a few months
 - now: days or hours
 - labor-intensive: too many new viruses
 - currently, 8-10 new viruses per day
 - can't handle epidemics:
 - queue of viruses to be analyzed overflows
 - heavy demand on server that posts signatures & fixes
- Automated analysis, e.g. “Immune System”
 - developed at IBM Research
 - licensed to Symantec

Immune System Architecture



Signature Extraction at VAC

- Virus allowed (encouraged) to replicate in controlled environment in immune center
- This yields collection of infected files
- In addition, a collection of “clean” files is available
- Machine learning techniques used to find strings that appear in most infected files and in few clean files, e.g.:
 - search files for candidate strings
 - add points if found in infected file
 - subtract points if found in clean file

Disinfection

- Once virus is detected, would like to clean up infected file
- AV tool must identify virus as well as disinfect file
 - can then supply code to remove the virus
 - requires detailed understanding of how the particular virus attaches

Macro-viruses

- Written in macro-language
- Infect documents (as opposed to programs), such as word-processor docs, spreadsheets, etc.
- “Attach” by modifying commonly used macros, or start-up macros
 - popular target is Normal.dot, which is opened when MS Office applications are executed
- Spread when documents are transmitted, via disks, file transfer, e-mail attachments, ...
- Macro virus dependencies:
 - Application popularity
 - Macro language depth
 - Macro implementation
 - Macro users

Virus Prevention

- Install high-quality Anti-Virus software and update signatures frequently
- Beware of executable files from untrusted sources (including innocent victims of virus attacks)
- Beware of macros

Hoaxes

- Warnings of non-existent virus attacks with instructions to warn everyone you know
- Can result in flood of e-mails, possible denial of service
- Check with reliable sources, such as CERT, before [usually = instead of] forwarding such warnings

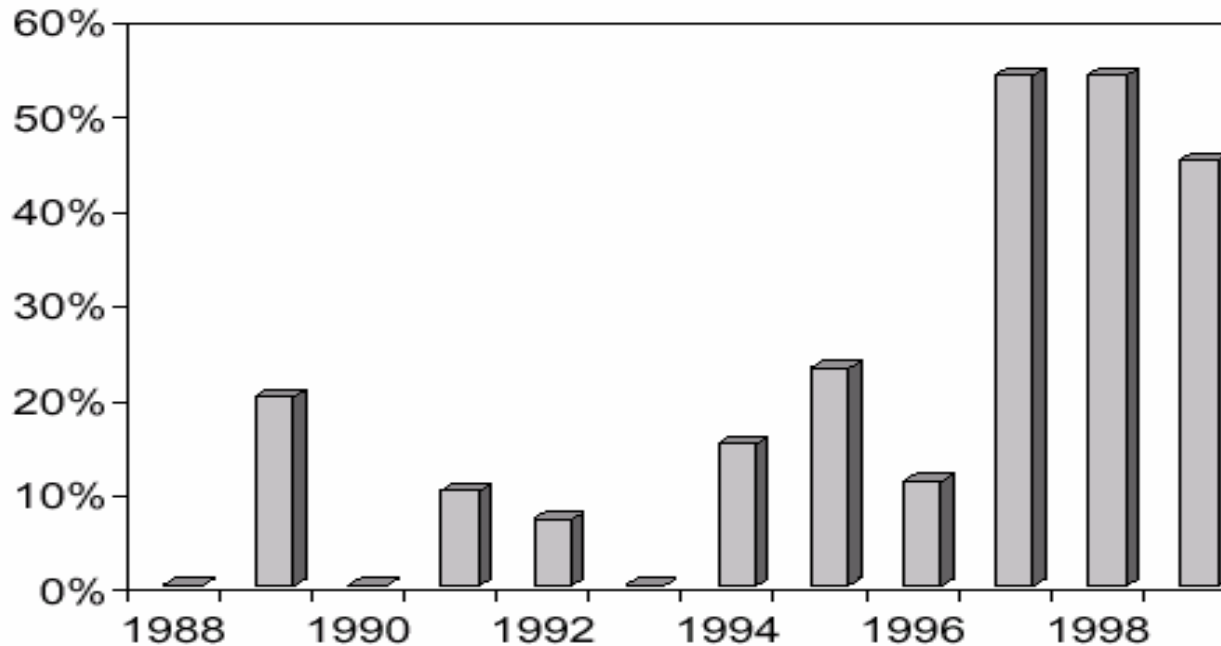
Means of exploiting a system

- User to root:
 - A local user on a system gets control of the system through privilege elevation.
- Remote to user:
 - An attacker on the network gains access to a user account on target host.
- Remote to root:
 - An attacker on the network gets control of the target host. No privilege elevation because attacker is not a user.
- Remote disk read:
 - An attacker on the network gains ability to read private data files on the target host without the authorization of the owner.
- Remote disk write:
 - An attacker on the network gains the ability to write to private data files on the target host without the authorization of the owner.

Buffer overflow

- Anecdotal notes suggest buffer overflows were known since sixties
- Take advantage of the lack of array bounds checking in C and C++ (and other languages) to transfer control to malicious code.
- Despite the fact that this vulnerability is well-known and preventable, buffer overflow attacks still prevalent

Buffer overflow



- Over 50% of the major security bugs leading to CERT Coordination Center advisories in 1999

Why is buffer overflow prevalent?

- **C/C++ is inherently unsafe?**
 - C/C++ library contains many “unsafe” functions
 - Pre, post conditions to these functions are not well understood
- **Lack of bound checking**
 - Programmers often forget to do bound checking
 - C/C++ don't do bound checking, unlike Java
 - “Unsafe” functions lack bound checking
- **Code reuse**
 - Many unsafe libraries are heavily reused e.g. libc

Buffer overflow

```
char sample[10];
```

```
sample[10] = `A`;
```

Or:

```
for (i=0; i<10; i++) sample[i] = `A`;
```

```
sample[i] = `B`
```

Can overflow into: user data, user program code, system data,
system program code

Leads to: computing with a faulty value or execution of an
improper/unintended instruction

General effects of buffer overflow

- No effect (if overwritten variables are not subsequently accessed)
- Program crash
- Strange output (possibly difficult to reproduce)
- Malicious effects
 - if buffer overflow is carefully exploited to transfer control to malicious code
- Note that buffer overflows may be difficult to detect with conventional software testing.

Exploiting buffer overflow

- Overflow a buffer allocated on the stack or on the heap in a way that causes the value of some important variable to change
 - e.g. flag indicating whether program can access private files
 - (harder w/ heap (dynamically allocated) allocated arrays, since memory map can't be predicted.)
- “Smash the stack”:
 - Overflow buffer allocated on the stack to change return pointer to the address of some malicious code

Smashing the stack

- Find a stack-allocated buffer such that overflow will allow return address to be overwritten.
 - Requires detailed understanding of activation record layout
 - Use debugger or modify code to dump relevant addresses and their contents
- Place hostile code in memory.
- Write over return address to cause jump to hostile code.

Popular Targets

- Processes that run with higher privileges
- Many UNIX functions need higher privileges to do things like writing to mail queue or opening a socket
 - Ordinary user programs may be temporarily granted higher privileges to invoke such functions
- Transfer control to code that spawns a shell
- Then use the shell (running with root privileges) to do whatever you want.

Buffer overflow defenses

- Do bound checking where necessary
 - The overhead is small compared to the risks
 - Write daemons in a type-safe language like Java
- Using dynamic memory(heap) for buffers does not solve the problem. Only makes it harder to exploit
- Integrate security into software developments
 - Code reviews
 - Testing
 - Checking security of added components

Buffer overflow defenses

- Non-executable buffers
 - Make data segment of address space non-executable. Although this is how older operating systems were designed newer versions of Unix and Windows make the data segment executable for performance optimization
 - Signal delivery
 - Trampoline functions
 - LISP functions
- Bound checking by compilers
 - Compilers can insert bound checking code at compile time e.g. Compaq C Compiler
 - Library functions are not checked
 - Code might be too complicated to automate

Buffer overflow defenses

- Type-safe Languages
 - If type-safe operations can be performed on a given variables then arbitrary changes cannot be used to execute code
- Stack integrity checks
 - StackGuard: Compiler generates integrity checks on activation records
 - PointGuard: Compiler generates integrity checks on instruction pointer

Guarding against malicious code

- Good software engineering practice
 - code reviews
 - careful independent testing
 - proof of correctness
- Operating system controls
 - restricted access
 - audit logs
 - etc

Further Reading

- Ludwig, *The Giant Black Book of Computer Viruses*, American Eagle Publications, 1995.
- McGraw, Viega, “Make your software behave: Brass tacks and smash attacks”, <http://www-4.ibm.com/software/developer/library/smash.html>
- White et al., “Anatomy of a Commercial-Grade Immune System” (and many other papers on viruses), <http://www.av.ibm.com/PapersFrame/papersframe.html>
- www.cert.org