

CMPUT 299 (B1) Winter 2008

# **Security in a Networked World**

*Malware*

yannis@cs.ualberta.ca

# References

- ▶ **Malware: Fighting Malicious Code**  
by Ed Skoudis with Lenny Zeltser
- ▶ **Web Hacking Attacks & Defense**  
by Stuart McClure, Saumil Shah, and Shreeraj Shah
- ▶ **Counter Hack Reloaded 2<sup>nd</sup> Ed.**  
by Ed Skoudis with Tom Liston

# Malware

- ▶ Malware: malicious code (usually *intended* to be malicious)
- ▶ Two types of classic malware: viruses and worms
- ▶ A basic observation: data can be executable instructions, there is no real separation of what constitutes data and what are instructions. The difference is merely of how one attempts to interpret them.
- ▶ Malware can propagate as data transfers where the data are a representation of executable code.
- ▶ Why is malware an important issue:
  - ▶ mainly due to the degree of connectivity, but also
  - ▶ due to the relative uniformity of platforms, and the
  - ▶ ever expanding user base.

# Types of Malware

- ▶ Viruses
- ▶ Worms
- ▶ Malicious mobile code
- ▶ Backdoors
- ▶ Trojan horses
- ▶ (user- & kernel-mode) Rootkits

# Viruses

- ▶ Virus: a self-replicating piece of code propagating mainly via human user actions.
- ▶ The first self-replicating code appears to have been PERVADE by John Walker (1975).
  - ▶ PERVADE was part of ANIMAL (a guessing game for animal names). PERVADE would infect directories with copies of ANIMAL (including PERVADE).
- ▶ First clearly malicious virus appears to have been the Elk Croner for Apple II (1982)
  - ▶ Elk Croner resided within boot sector of floppies, and copied itself to new disks and every now and then would display an irritating message (in verse!).

## Viruses (cont'd)

- ▶ The first virus to attack Microsoft DOS appears to have been “Brain” (1986) which infected the boot sector, and, about the same time Virdem (1986) from the Chaos Computer Club which infected .COM archives.
- ▶ The fundamental problem of a virus is that it has to find a host which is going to be (sooner or later) executed. The boot sectors of floppies was a natural choice as long as people would boot from floppies. Today viruses rely on attaching themselves either to some executable file, or to the master boot record (MBR).

# Virus Infection Mechanisms

- ▶ Attachment to an already executable file. In Windows these are .EXE and .COM files. .EXE follows the Portable Executable (PE) format which is also used by .SYS, .DLL, .OCX, .CPL, and .SRC files.
- ▶ Occasionally viruses have been installed as OS drivers, like the Infis virus which appeared as an NT/2000 kernel-mode driver.
- ▶ How is attachment achieved:
  - ▶ With modification of the executable file or
  - ▶ Without (!)

## Infection Without Modification

- ▶ Exploit the UI limitations and the fact that the executable to run depends on the order it appears in a user's PATH.
  - ▶ `program.com` precedes `program.exe` and can run when the user enters `program`. Also, `program.com` can be hidden (from directory listings).
- ▶ Modification of the filename of the normal program and assumption of that program's name.
  - ▶ Rename `program.exe` to `program.ex_`
- ▶ Note that control can eventually pass to the “infected” program to avoid detection.

## Infection Without Modification (cont'd)

- ▶ On occasion, file systems can be a place to hide.
  - ▶ For example: NTFS provides for *alternate data streams (ADS)*.
    - ▶ Each file can use a primary/default or an alternate data stream.
    - ▶ Infection can add an alternate data stream, point to it as the default and never modify the primary.
    - ▶ An example is the Win2K.Stream virus

## Infection Through Modification

- ▶ Prepend (used mostly in .COM but also in .EXE, e.g. like the Nimda worm did), append (like Bliss in Linux), or (partially?) overwrite an existing executable file.
- ▶ .VBS, .PHP, etc. files have also been victims of infection. The Appix worm (2002) prepended to .COM, .EXE and .SCR and appended to PHP.
- ▶ Boot Sector modification: runs before the system is booted by moving the contents of the MBR someplace and replacing them with the malicious code which eventually might pass control to the original MBR, e.g., Michalangelo (1991).

# Infecting Documents

- ▶ Any file format that provides options for macros, or some other user-programmable behavior, is a potential virus vector.
  - ▶ Microsoft Office (VBA, VB .NET, C# .NET)
  - ▶ Wordperfect Office (VBA, PerfectScript, ObjectPAL)
  - ▶ StarOffice/OpenOffice (StarOffice Basic)
  - ▶ AutoCAD (VBA)
  - ▶ ...
- ▶ Special function names are called at certain events, e.g., opening or closing a file. The Melissa (1999) virus used this feature. The virus can copy itself to template files that persist.

## Infecting Documents (cont'd)

- ▶ Really awesome *multipartite* virus: Navrah
  - ▶ Infect using Word documents,
  - ▶ Extract malicious code from document, to
  - ▶ Infect device drivers.
  - ▶ On reboot, Navrah is loaded into memory
  - ▶ The sky is the limit ...
- ▶ Multipartite: the infection is to various elements of the system. (Usually executables and boot sectors.)

## More Infection Targets

- ▶ Any interpreted scripting language.
- ▶ Source code (!)
  - ▶ Rare, but existing. ScrVir and Urphin are examples.
- ▶ Java Code
  - ▶ StrangeBrew (1998) write once run everywhere :-)
  - ▶ But the Java Virtual Machine JVM limits the potential of harm that can be done (but JVMs have on occasion been attacked).
- ▶ Propagation is usually via removable media, by e-mail, downloading, and shared filesystems.

# Defenses

- ▶ Anti-virus software mainly operates on pattern matching for virus “fingerprints”
- ▶ System files are to be checked (generating strong hashes) and verified against a R/O source of authoritative hashes.
- ▶ Enforce the principle of *least privileges* and disable *non-essential active components*.

# Virus Self-Defense

- ▶ Stealth
- ▶ Polymorphism
  - ▶ In essence, the virus encrypts itself (possibly with a different key each time) and self-decrypts when it needs to become active. Encryption includes the possibility of “scrambling”/reordering the code to avoid exhibiting the exact same pattern.
- ▶ Metamorphism
  - ▶ A virus “re-programs” itself by replacing/exchanging instructions, adding superfluous instructions, etc., but without violating the logic being implemented.
- ▶ Evading Anti-Virus Tools

# Worms

- ▶ Worm: malicious software that spreads without user intervention.
- ▶ Current theoretical studies indicate that a really infectious worm could spread in a matter of less of an hour to the entire Internet.
- ▶ The basic problem is the fast determination of *vulnerability* of a victim host. If the process can be performed in advance, then the worm can be informed of which area of the IP address space to attack.

# A Quick History of Famous Worms

- ▶ Morris Worm (1988) [Unix]
- ▶ Melissa (1999) [MS Outlook]
- ▶ Code Red (2001) [IIS Server]
- ▶ Nimda (2001) [IE, IIS, file sharing, MS Outlook]
- ▶ SQL Slammer (2003) [MS SQL Server]
- ▶ ...

# Structure of a Worm

- ▶ Warhead
- ▶ Propagation Engine
- ▶ Target Selection Logic
- ▶ Scanning Engine
- ▶ Payload

# Warhead

- ▶ Buffer Overflow Exploits
- ▶ File Sharing
- ▶ E-mail
- ▶ All kinds of other system vulnerabilities

# Propagation & Target Selection Mechanisms

- ▶ When the worm is significantly large, and it might not be able to be transferred along with the warhead, the warhead “pulls” the rest of the worm using a file transfer/sharing protocol:
  - ▶ FTP, TFTP, HTTP, SMB, etc.
- ▶ Target selection is based on harvesting information about the user and its working environment.
  - ▶ E-mail addresses
  - ▶ IP subnets / DNS entries
  - ▶ Trusted systems and host lists
  - ▶ Even randomly!

# Scanning Engine

- ▶ With targets selected, their vulnerability has to be assessed.
  - ▶ Vulnerability is usually just the test whether the “warhead” part is going to be successful or not.
- ▶ The main problem with scanning (at least for a virus writer) is that, after a point, a scanned host may in fact be already infected.
  - ▶ Counter-productive scanning, usually manifests as an infection slowdown once a critical number of hosts have been infected.
  - ▶ But it is no longer uncommon for a worm that detects the presence of another worm, to exploit the other worms' existence (e.g. backdoor) and replace it.

# Payload

- ▶ The payload contains the non-propagation logic part of the worm. This logic can be absent (worms just for the fun of propagation) or be any executable code intended to be used for:
  - ▶ Backdoor installation.
    - ▶ Appropriately hidden like Trojan Horses, RootKits etc.
  - ▶ Zombification.
    - ▶ Adds infected host to the minions of a Distributed DoS
  - ▶ Resource harvesting.
    - ▶ For example CPU cycles for large distributed computation.
  - ▶ ...

## Nimda (2001)

- ▶ An assortment of warhead exploits:
  - ▶ IIS not checking properly paths in HTTP requests
  - ▶ Browsers downloading worm from infected IIS
  - ▶ Outlook (pre)viewing of an infected message
  - ▶ Attached to .exe and .html/.asp files on shares
  - ▶ Reused (!) backdoors from Code Red II
- ▶ Propagation via HTTP, SMTP, SMB & TFTP
- ▶ Target Selection
  - ▶ E-mail addresses (addressbook & html documents)
  - ▶ Preferred generation of “nearby” IP addresses

## Nimda (2001) (cont'd)

### ▶ Payload

- ▶ Apparently intended to leave the system “naked”
  - ▶ Allowed file sharing of the primary hard drive.
  - ▶ Activated the Guest account.
  - ▶ Added Guest account to Administrator group/role.

### ▶ Life beyond Nimda

- ▶ Multiplatform worms (“multi” is not that broad anyway)
- ▶ Multiexploit worms
- ▶ 0-day exploit worms
- ▶ Accelerated spreading worms
- ▶ Polymorphic & Metamorphic worms

# Defending Against Worms

- ▶ The more elaborate the worm, the larger the data transfers needed to propagate it. They can hardly fly “below” the radar of Intrusion Detection Systems. (but SQL Slammer is an example of a “lightweight” worm that used just UDP packets.)
- ▶ The tools of the trade
  - ▶ Antivirus tools (but next to useless with 0-days)
  - ▶ Patches (lots of them)
  - ▶ Firewalls & IDSes & monitoring/incident response
  - ▶ ...
  - ▶ Ultimately, it's an arms race.

# The Gift that Keeps on Giving: The Web

- ▶ Vulnerabilities related to web applications are the norm in today's networked world. These vulnerabilities are typically due to:
  - ▶ Vulnerabilities of the server
  - ▶ Vulnerabilities of the server backends
  - ▶ Vulnerabilities of the client (browser)
  - ▶ Vulnerabilities due to delivered content
- ▶ To some, the biggest menace is the freedom with which remote content can be executed locally at the client ("mobile code").
  - ▶ Usually JavaScript and ActiveX.
    - ▶ Note JavaScript  $\neq$  Java

## JavaScript (scratching the surface)

- ▶ Allows actions upon certain events (e.g., page loading or unloading).
  - ▶ Easily used for DoS, e.g., `onload()`
- ▶ Interferes with the UI's (legitimate) visual representation of other information
  - ▶ Can occlude the presentation of other info
- ▶ All kinds of irritants (redirections, opening other windows, etc.),
  - ▶ Check [www.guninski.com/browsers.html](http://www.guninski.com/browsers.html) for some
- ▶ ActiveX is “worst” in some sense (the code executes natively) but at least it provides the option of signing code.

## Statelessness (again)

- ▶ Cookies are information implanted by the server to a “cookie jar” at the browser for session identification.
- ▶ Cookies usually carry access rights, because the server does not want to store them locally.
  - ▶ Acquiring the cookie means acquiring the rights.
- ▶ Cookies usually expire (a good thing) but are accessible, in principle, by any other server.
- ▶ So, restrictions are placed on cookies as to which domain can access them.
  - ▶ But the restriction is not necessarily strong (think of the ways to tinker with DNS) plus buggy code ...

# URL Parsing Bugs

- ▶ Example: Bennet Haselton's discovery of IE 5.01 bug
- ▶ The browser identifies the domain of the server by parsing the URL.
- ▶ An attacker composes a URL that appears to be from the domain allowed to access the cookie.
  - ▶ Normal URL (would force browser not to provide cookie)
    - ▶ <http://somattacker.ca/mypage.html?.legitdomain.com>
  - ▶ URL to trick parsing
    - ▶ <http://someattacker.ca%2fmypage.html%3f.legitdomain.com>
  - ▶ Note the parser finds no / after the // and so the entire URL is treated as a domain name, and sure enough it is (incorrectly) found to be from within the .legitdomain.com
  - ▶ Alternate/escape character representations (%2f, %3f, etc.) needs to be accounted for by the parser.

# XSS (Cross-Site Scripting)

- ▶ An attacker enters (e.g., in a form entry) JavaScript code. The form was meant for text, or for something innocuous anyway but it is now “infected” with JavaScript
- ▶ An unsuspecting user's browser loads the page which presents the contents loaded by the attacker, and hence loads the JavaScript code.
- ▶ The executed JavaScript can access whatever a locally executing JavaScript is allowed to access (might depend on configuration) but usually has access to the cookies related to the server.
- ▶ The JavaScript code sends the obtained info/cookies to the attacker's server. If the attacker obtains the cookies, he/she may be able to impersonate the victim.

# More Web-Related Concerns

- ▶ HTTP Vulnerabilities
  - ▶ Cookie
  - ▶ User-Agent
  - ▶ Referer
  - ▶ Accept
- ▶ Server Side Vulnerabilities
  - ▶ Basic & Digest Authentication Modes are weak
  - ▶ Server-Side Scripts, e.g., PHP
  - ▶ And an interesting challenge: where to place a web server with respect to a firewall. It should be accessible to the “public” after all, right? (DMZ?)

# Tainted Input

- ▶ Any opportunity for a user to provide input is a cause of concern for badly written code.

- ▶ Example:

```
$mystring = "/usr/ucb/mail $addr < /tmp/somedoc";  
system("$mystring");
```

An interesting tradeoff: sanitizing the input (i.e., removing certain “dangerous” elements (; .. etc.)) appears essential but also implies server work. A server can either impose a very specific input form (deny all except those that comply) rather than allowing a general one and then cleaning up the mess (accept all except those that include dangerous elements). [This is a recurring theme in security.]

# Tainted URLs

- ▶ URL format:
- ▶ **protocol://server/can/be/long/path/resource?params**
- ▶ The path together with the resource *usually* map to a directory path and corresponding script/program.
- ▶ Technically, the components of the path can be interpreted via an arbitrary mapping performed by the web server in order to locate a resource, e.g. they can be “aliased”.
- ▶ Example multiple parameters:
  - ▶ **item=MSk2243&payment=paypal**
  - ▶ also '+' is interpreted as space

# Input Validation

- ▶ The source of many security problems for web servers is insufficient input validation, and often URL validation.
- ▶ A malformed URL can cause a crash of the application/resource handling it (e.g., replace numerical values with strings, or small numerical values with large ones!).
- ▶ Reserved characters can be passed using 2-digit hex values after a '%' “escape” character.
  - ▶ %0a is a linefeed, %0d is a carriage return, %26 is the ampersand, etc.
- ▶ Many servers recognize 16bit UTF (%uXXXX)



# Unicode Vulnerabilities

- ▶ A single character (e.g., '/') can have multiple encodings according to UTF-8 multi-byte representation modes:
  - ▶ '/' = 0x2F, or 0xC0 0xAF, or 0xE0 0x80 0xAF
- ▶ Any software that interprets multi-byte UTF must:  
*“... not accept UTF-8 sequences that are longer than necessary to encode a character. Any overlong UTF-8 sequence could be abused to bypass UTF-8 substring tests that look only for the shortest possible encoding.”*
- ▶ The rule was not followed in certain IIS versions.

# Multi-byte UTF-8 Attack

## Example:

- ▶ `http://192.168.7.21/scripts/..%c0%af../winnt/system32/cmd.exe?/c+dir+d:\`
- ▶ First the check for directory accessibility identifies that the above URL is not looking anywhere outside the `C:\inetpub` where usually the web pages of ISS were placed. “`..%c0%af...`” appears like a regular, albeit oddly named subdirectory
- ▶ Then the path is interpreted (improperly) as a multi-byte UTF-8 name, and essentially translated to
- ▶ `http://192.168.7.21/scripts/../../../../winnt/system32/cmd.exe?/c+dir+d:\`
- ▶ What is then accessed is effectively `c:/winnt/system32/cmd.exe`

# Double/Superfluous Decoding

## Example:

- ▶ `http://192.168.7.21/scripts/..%25%32%66../winnt/system32/cmd.exe?/c+dir+d:\`
- ▶ In an attempt to check whether there is anything “fishy” (like slashes) with the input, we replace the hex encoding characters with their ASCII representation. The output looks OK (no slashes), but note that `%25=%`, `%32=a`, `%66=f`, hence:
- ▶ `http://192.168.7.21/scripts/..%af../winnt/system32/cmd.exe?/c+dir+d:\`
- ▶ Which is again:
- ▶ `http://192.168.7.21/scripts/../../../../winnt/system32/cmd.exe?/c+dir+d:\`
- ▶ What is then accessed is effectively `c:/winnt/system32/cmd.exe`

## Backends

- ▶ Usually web application front ends access a backend database. The language of choice for database access/manipulation is SQL.
- ▶ The Catch 22: in many cases user input is used to form an SQL query (otherwise the services provided are not particularly flexible).
- ▶ The user input is used as a source of malicious SQL in what is termed an “SQL injection” attack.
- ▶ The database itself may be located behind firewalls, but access to it is facilitated by the web server that is accessible (even if behind the firewall, e.g., in a DMZ).

## Examples of Input and Lack of Validation

- ▶ `http://192.168.0.6/purchase?ID=1`
- ▶ If the above generated an output that seems to have come from a database, one can reasonably assume that ID is a mapped to a field/attribute in the database.
- ▶ The ID parameter is passed to the database (maybe via some transformation).
- ▶ So let us craft the following URL:  
`http://192.168.0.6/purchase?ID=2'%20OR%20'1'='1`
- ▶ A look into a plausible SQL snippet is due

## The SQL Query

- ▶ A likely SQL query candidate:

```
SELECT * FROM inventory WHERE  
    inventoryID = '$ID';
```

- ▶ Hence the result of the SQL injection is:

```
SELECT * FROM inventory WHERE  
    inventoryID = '2' OR '1'='1';
```

(which of course will retrieve all entries of *inventory*)

## Schemas Revealed

Through interaction with the regular input one can determine with reasonable accuracy the kinds of information present in the DB schema.  
(Guesswork.)

```
http://192.168.0.6/purchase?ID=2'%20AND  
%20owner%20IS%20NULL;%20--
```

Leading to:

```
SELECT * FROM inventory WHERE  
inventoryID = '2' AND owner IS NULL; -- ' ;
```

# Backdoors

- ▶ Backdoors are usually installed as part of a worm's penetration to allow future access.
  - ▶ A backdoor is usually a “server” which frequently allows direct or indirect access to a command shell.
- ▶ Intrusion detection tools and host-firewalls are usually sensitive to servers starting up for no good reason.
- ▶ An alternative is to avoid starting a server and instead start a client (!).
  - ▶ Such is the strategy of reverse-tunneling tools.

## Example: Reverse HTTP Tunnel

- ▶ A compromised system runs a backdoor process that sends HTTP requests (technically indistinguishable from any other outbound HTTP request, e.g. to port 80) to a particular server.
- ▶ The server being contacted is under the attackers control. In fact, it is not even a “true” HTTP server but HTTP syntax is used to disguise the purpose of the communication.
- ▶ The server responds by sending back commands to be executed by the client program and the client program can send back responses disguised as HTTP requests.
- ▶ The only problem is that there should be a “rendezvous” for the two components to meet. Usually, the client is activated on a periodic basis.

## Other Backdoors Without Ports

- ▶ ICMP Backdoors (Loki, 007shell, etc.)
  - ▶ ICMP does not use ports
  - ▶ (some) ICMPs are usually allowed through firewalls
  - ▶ ICMP messages have lots of “vacant payload space”
  - ▶ Same req./response strategy as with reverse HTTP
  - ▶ Drawback: unreliable transfer (but who cares)
- ▶ Non-promiscuous Sniffing Backdoors (Cd00r)
  - ▶ A non-promiscuous sniffer keeps track of packets sent to a particular sequence of ports and considers them a signature of the master.
  - ▶ Many variations of the basic idea exist.
  - ▶ The basic principle is that of a covert channel.

# Covert Channels

*Covert channels are shared resources as paths of communication.* (Bishop)

- ▶ Use an innocuous publicly observable action (or side-effect of an action) as a means of sending information. The meaning of the action (or its side-effects) are known only to the two communicating entities.
- ▶ Example: closed window is 0, open window is 1. Observed periodically, say once a day at a particular time. (Q: What was the covert channel used in X-Files?)
- ▶ Shortcomings: covert channels usually suffer from low bandwidth, and tend to be noisy. But the meaning of a single bit of information depends on the arrangements made by the two communicating entities.

# Backdoors and Covert Channels

- ▶ The use of a covert channel in a backdoor scenario is mainly for “signaling” purposes, that is to trigger/activate some backdoor functionality.
- ▶ The data content to be transferred is usually too much to rely just on a covert channel.
- ▶ On the other hand, if quick response is not an issue, the response can be also coded into a low rate covert channel.
- ▶ Intrusion detection systems (IDS) usually have a threshold of events before they announce some activity as suspicious. The threshold tends to be high to avoid false positive threat identification (which subsequently requires human attention). Hence, a low rate covert channel can “fly under the radar” of an IDS's tolerance.

# Promiscuous Sniffing Backdoors

- ▶ Commands are sent to a host on the same LAN/subnet as the one in which the backdoor is installed. The backdoor obliges the same way as it would in the Non-Promiscuous Sniffing Case.
- ▶ What this strategy buys the attacker is the apparent lack of correlation between the host receiving the control signals and the host that responds.
- ▶ It can also be used to misdirect efforts for patching to the systems where the backdoor is not really present.
- ▶ A more tangled prospect is that of using many intermediate hops (on different LANs) to forward commands and responses. Seemingly correlated with the regular traffic.

## Things That Fall Into Place

- ▶ You should now be able to recognize why security suites (antivirus+firewall software) care about:
  - ▶ Processes listening to certain ports (servers).
  - ▶ Clients accessing hosts known to be not trustworthy.
  - ▶ Executables that are unknown and not trustworthy.
- ▶ Invariably, modern security suites ask user confirmation for each of the above cases.
  - ▶ The downside is that in many cases the answer is to accept the suspicious behavior.
  - ▶ From a User Interface perspective it becomes difficult to avoid users conditioned to clicking OK all the time.

## Mitigation of Covert Channels

- ▶ The shared resources are the channel, and the encoding is possible by “modulating”/changing the shared resource. From an OS perspective:
- ▶ Strategy A (Isolation): Force processes to declare their resources in advance, and book them for the amount of time the processes run.
- ▶ Strategy B (Obfuscation): Introduce sufficiently high levels of noise such that the covert channel throughput rate diminishes to practically zero.
- ▶ Both approaches are at odds with efficiency.

# A Digression: An Opinion about Security Suites

- ▶ Security tools today are heavily relying on “signatures” and pattern matching algorithms.
- ▶ This works OK for known threats, but practically useless for 0-day threats and threats with crafty poly/meta-morphic character.
- ▶ Hence, many tools offer run-time behavior checking which requires user feedback to an irritating degree.
- ▶ Run-time behavior checking raises the following distinct paradox (think e.g., of JavaScript code) to determine the run-time behavior of a suspect code/system *we have to run it before we run it*.
- ▶ In the age of abundance of CPU & memory, it might look like a good deal after all, but one cannot help but think whether we can do this better.

# The Kings of Malware: Rootkits

- ▶ User-mode rootkits:
  - ▶ Altering the tools provided by an operating system (executables & libraries) to allow backdoors to be installed and hidden from view.
  - ▶ Example: change the ps command to skip certain processes from being displayed.
- ▶ Kernel-mode rootkits:
  - ▶ Modification of OS kernel functional units, such that the user-level tools need not be modified, but still provide the ability to store backdoors and hide them from view.
  - ▶ Example: alter the system calls by which the list of processes can be acquired.

# Usual Arsenal for User-Mode Rootkit

## ▶ Linux/Unix (lrk6)

Executables: chfn chsh crontab du find ifconfig inetd  
killall login ls netstat passwd pidof ps rshd syslogd  
tcpd top sshd su

## ▶ Windows

Much harder due to the Windows File Protection to replace a system file BUT much easier to attach a process to another process (normally used for debugging) and modify it by “API hooking” the APIs to hook on are the standard APIs provided by handful of Windows DLLs

Example: hxdef (by the “holy father” !)

## Kernel-Mode Rootkits

- ▶ One potent technique is that of execution redirection. An executable indicated to be executed is in fact not the one that gets executed!
- ▶ Loading and execution of new processes is controlled via specific OS system calls. Such calls can be changed to execute the compromised version of a program.
- ▶ The hidden version of the program is hidden by compromising the system calls that provide information about files & directories.
- ▶ Example: Adore-ng (for Linux) FU (for Windows)

# Buffer & Heap Overflows

- ▶ Technically, they are all attempts to exploit insufficient runtime checks with the intent of subverting the control flow of a program.
- ▶ Buffer overflows:
  - ▶ Overwrite the return address adjacent to the the local/ auto area in the stack frame of a subroutine.
- ▶ Heap overflows:
  - ▶ Overwrite global variables, and in particular pointers to crucial data or function pointers to subroutines.
- ▶ Both attacks leverage the high degree of *predictability* of the state of the stack and/or of other locations inside a process logical address space.
- ▶ Executable code can be injected disguised as data (a user supplied buffered input), as standard arguments (e.g. command line), etc.

# Languages and Security

- ▶ What is wrong with C ?
  - ▶ Pointer arithmetic.
  - ▶ Liberal use of pointer casting.
  - ▶ Lack of bounds checks.
  - ▶ Certain libc functions.
- ▶ C heavily relies on the programmer understanding the runtime behavior of a program.
  - ▶ This is a difficult skill to acquire. Also,
  - ▶ static checks, e.g., during compilation are insufficient.
  - ▶ some checks are inherently doable only at run-time
    - ▶ e.g., bounds checks.

# Java

- ▶ Java is one of the few languages that include the notion of a security policy.
- ▶ The policies are expressed in `java.policy`
- ▶ Examples:

```
// If the code is signed by "Duke", grant it read/write access to all
// files in /tmp:
grant signedBy "Duke" {
    permission java.io.FilePermission "/tmp/*", "read,write";
};
// Grant everyone the following permission:
grant { permission java.util.PropertyPermission "java.vendor"; };
```

(<http://java.sun.com/j2se/1.3/docs/guide/security/PolicyFiles.html#Examples>)

# Java Language Security

- ▶ Java is designed to be type-safe.
- ▶ Strict adherence to visibility rules (`private`).
- ▶ Bounds checks.
- ▶ Variables must be initialized to be used.
- ▶ `final` is adhered to.
- ▶ Casting limited to superclass  
(or to subclass if the object is an instance of the subclass).
- ▶ No pointers or access to arbitrary memory.
- ▶ Automatic memory management, garbage collection (no explicit memory allocation).

# Java Bytecode Verifier

- ▶ Bytecode verifier: only legitimate Java bytecodes are executed. The bytecode verifier, together with the Java Virtual Machine, guarantees language safety at run time.
- ▶ Examples
  - ▶ control-flow instructions go to the start of an instruction,
  - ▶ register references are to a legal register (the method indicated how many),
  - ▶ stack manipulated only as allowed by a given instruction, etc.

# Java Classloader

- ▶ Classloader: (over-simplified) provides a local namespace such that remote code is not allowed to interfere with the running of other code.
  - ▶ Example: same class name from different sources results in different “effective names” when loaded locally. Hence a class is intrinsically linked to the classloader instance that brought it over. (For browsers: applets with different codebase, even if originating from the same host, are loaded by different instances of the browser's class loader.)
- ▶ Since Java 2, there is no notion of a particular sandbox. Rather, the security policy determines (with fine-grained access control) the access rights of the executing application.

## Trust (again)

- ▶ In Java we fundamentally trust the correct operation of JVM. Yet JVM is often written in C!
- ▶ Broader issue: do we trust the code produced by others? [The mechanical facet of trust, i.e., signed code, is not sufficient.]
- ▶ The code (in its source form anyway) may be trusted, but do we trust the outcome of compilation? [Can the compiler sign the code it produces? Then how do we trust the compiler that compiled the compiler?]
- ▶ Ken Thompson's "Reflections on Trusting Trust" (ACM Turing Award Lecture)

# Trusted Computing Base (TCB)

- ▶ An environment (hardware+software) that provides security guarantees.
  - ▶ Usual checks and balances: is the hardware to be trusted? Is the boot process secured? Are the applications trustworthy? Etc.
- ▶ Even small commodity platforms, e.g. Xbox, attempt to implement a TC platform.
  - ▶ Yet it took a broken “certified” application, to load a Linux kernel without any hardware modification (via the saved data of the trusted game).
- ▶ A “true” TCB requires that all hardware and code is proven mathematically to be correct.