Computer Security

06r. Malware: review & defense
Assignment 5 review

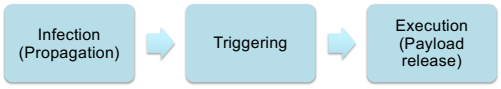
Paul Krzyzanowski • David Domingo • Ananya Jana
Rutgers University
Spring 2019

Worms vs. Viruses

- Conceptually similar
  - Software that replicates itself onto other systems
    - May be spread automatically (via network access) or manually (e.g., email attachments, flash drives)
  - Key distinction is whether they are standalone
- Worm
  - Standalone software
- Virus
  - Requires a host program: a virus attaches itself to another piece of software

Virus components

- Infection mechanism
  - Search for infection targets: other programs, specific files, disk areas
- Payload
  - The malicious part of the virus
- Trigger (logic bomb)
  - Executed whenever a file containing the virus is run
  - Determines whether the *payload* should be delivered
    - Virus may stay dormant for some time



Infiltration mechanisms: summary

Some ways in which malware enters a system

How does malware get onto a computer?

- You installed it
  - Social engineering
    - You were fooled into install software or click on something that triggered the installation: e.g., "System cleaner" software, software "updates", cracked versions of expensive software, license key generators, …
    - Phishing attacks: usually email that is meant to look legitimate but contains a malicious attachment or link
    - Spear phishing attacks: personally targeted email meant to look legitimate
  - You thought you were installing something else
    - File sharing downloads, crack generators
  - Embedded macros: your document or spreadsheet executed code
- Infected removable media
  - USB drives with malicious firmware, setup programs
- Hacking: attacks on services running on the computer

Residence: summary

Some ways in which malware lives in system

Finding a home

- Files & startup scripts
  - Worms can live as commands and launched at system boot
- File infector viruses
  - Malware that adds itself to a legitimate program without the user's knowledge
- Macros
  - Like file infectors but attached to documents rather than programs
- Bootloader & firmware viruses
  - Boot loaders or system firmware can reinstall malware each time the system is booted
- Hacked source repositories

Finding a home

- Rootkits
  - Modifications to commands or the kernel to hide the presence of malware
  - An administrator won't see the malicious files or processes
- Hypervisor rootkits
  - Malicious hypervisor software can intercept all interactions between the operating system and devices
- File-less malware
  - Process lives only in memory
  - Evades anti-virus file scanning software; may be launched from rscripts run via Windows registry settings
- Trojan horses
  - Users think they're installing (or clicking) something legitimate
    - Overt purpose: known to a user – the legitimate part of the program ("it's a game")
    - Covert purpose: unknown to a user – the malware ("it's a cryptocurrency miner")

Operation: summary

Some things malware can do

Malware functions

- Spyware
  - Monitor browsing history, messages, files, camera/microphone
- Adware
  - Present ads to a user and/or generate clicks
- Ransomware
  - Encrypt files an demand ransom to decrypt
- Backdoors
  - Modify system settings or server software to create a way to access to software or a system that bypasses normal authentication mechanisms

Defenses

File Protection

- Embedded devices & older Microsoft Windows systems
  - User processes ran with full admin powers
  - This made it incredibly easy to install malware – even kernel drivers
  - Still a problem with most embedded devices (routers, printers, ...)
- Lack of file protection makes it easier to spread viruses
  - But it can be a pain even if only your files are affected
  - Viruses can override DAC permissions
- Warning users
  - Today's systems warn users about requests for installation or elevated privileges
  - For Trojans, many users will enter their password and say "yes" – they think they want the software
- MAC permissions
  - Can stop some viruses if users cannot install or override executable files
  - But macro viruses can still be a problem

Anti-virus software

- No way to recognize all possible viruses
- Anti-virus programs look for patterns of known viruses
  - "signature scanning"
- Viruses can defend themselves
  - Encryption: encrypt most of the virus – decrypt on execution
    - Only pattern is the decryption code
  - Polymorphic viruses:
    - Mutates the code each time it runs but keeps the algorithm the same
    - Viruses will encrypt their code but use polymorphism for the decryption code
- Sandboxing
  - Anti-virus software can run suspected code in a sandbox – or interpreted environment – and see what it tries to do
- Anomaly detection
  - Look for abnormal-looking behavior patterns

Anti-virus (AV) software

No way to recognize all possible viruses

Two main approaches
1. Signature-based
2. Behavior-based

- Signature-based systems
  - Anti-malware companies collect malware
    - Often study software in sandboxed environments to see what it tries to do
  - **Signature** = set of bytes that are considered to be unique to the malware
  - Signature scanning:
    - Presence of those bytes in a file tells us the code as malicious

Anti-virus software: Behavior-based

- Monitor process activity and stop the process if it is deemed malicious
- Sandboxing
  - Anti-virus software can run suspected code in a sandbox – or interpreted environment – and see what it tries to do
- Anomaly detection
  - Look for abnormal-looking behavior patterns

Behavior-based detection tends to have much higher false positive rates

Most AV products use signature-based detection

Defeating signatures

Viruses can defend themselves

- Encryption: encrypt most of the virus – decrypt on execution
  - Only pattern we can detect is the decryption code
- Pack the code – unpatch during execution
  - Need run-time detection or else use a signature of the packer
  - Packers compress, encrypt, or simply *xor* the payload with a pattern.
- Polymorphic viruses:
  - Modify the code but keep it functionally equivalent
  - Add NOPs, use equivalent instruction sequences
  - This changes the signature
  - Do this each time the code propagates

Better yet…
- Write your own malware.
- Maybe you can get away with just writing a packer.

Defeating signatures

- Detection requires scanning incoming data streams
  - But they can be encrypted
- Malware via HTTP/SMTP content
  - Admins often set up black lists for SMTP attachments and HTTP content
  - Blacklisting = list of disallowed content
    - E.g., people might disallow windows EXE files.
  - Whitelisting = list of allowed content
  - White lists are preferable it harder to manage
    - There could be a huge number of acceptable file types.
    - Similarly, black lists are dangerous since there are many formats that could transport executable files.
    - Microsoft lists 25 file formats that can be directly executable by double clicking
  - Attackers can exploit bugs in allowable content, such as PDF or Excel files

Defeating signatures

- Social engineering-based defeats
  - The attacker can pick an arbitrary format and use social engineering to ask a user to rename it.
  - Executable malware can also be embedded directly into Microsoft Office documents as an object. You then have to get users to click on it.

Removing admin rights helps a lot

From the Feb 25, 2017 Avecto Microsoft Vulnerabilities Report

- 530 Microsoft vulnerabilities reported in 2016
  - 94% of them could be mitigated by removing admin rights
  - 100% of vulnerabilities impacting IE and Edge could be mitigated by removing admin rights
- Breakdown
  - Windows 10 had 395 vulnerabilities
  - Window 8 & 8.1 had 265 each
  - Office was hit with 79 vulnerabilities
    - Removing admin rights
      would mitigate 99% of vulnerabilities in older versions
      would remove 100% of vulnerabilities in Office 2016

Note: the analysis only covers *known* vulnerabilities

http://www.computerworld.com/article/3173246/security/94-of-microsoft-vulnerabilities-can-be-easily-mitigated.html

Solving the problem

- Access controls don't stop the problem
- Privilege escalation limiting mechanisms work better
  - Removing admin rights is great … but user files remain at risk
  - Containment mechanisms (like containers) work well for servers
    - But not for end-user software
- Running software in a sandbox is great
  - Mobile phones rely on this
  - Often too restrictive for computers
  - You have to trust that users won't be convinced to grant the wrong access rights
- Trojans/worms that exploit human behavior are hard to prevent
  - We're dealing with human nature
  - We're used to accepting a pop-up message and entering a password
  - Better detection in browsers & mail clients helps … but risks junking legitimate content
- Simple software – without automatically-run macros is also good
  - *vi* vs. *MS-Word* … but isn't acceptable to a lot of users

It's still a big problem

Assignment 5 Review

Question 1 (a, b)

Ken Thompson's *Reflections on Trusting Trust* paper

(a) What does the match for *pattern1* test?

It matches the part of the login program that validates a user's identity.

(b) What does "bug 1" do?

Bug 1 detects a compilation of the login program and inserts code to allow bypassing a password check.

This allows the compiler to detect that it is compiling the *login* program (which is run to allow users to log into a system).

The compiler then adds code to bypass the password check.

⇒ You can examine the source code to *login.c* and see nothing suspicious!

Question 1 (a, b)

Ken Thompson's *Reflections on Trusting Trust* paper

(c) What is the purpose of the second pattern match in the C compiler – the one that inserts "bug 2"?

Bug 2 matches a pattern that detects the compilation of the C compiler

(d) What does "bug 2" accomplish?

(d) Bug 2 inserts BOTH Trojan horses into the compiler so the source does not contain any code that matches patterns in either the C compiler or the login program.

Question 1 discussion

- If the C compiler detects that it is compiling the C compiler, it will add:
  - The malicious code to the C compiler to detect the compilation of the C compiler and insert this code
  - The malicious code to detect the compilation of login.c and insert bug 1
- After you compile the C compiler, you can remove the detection/bug code from the source and it would be magically re-inserted each time you recompile the compiler

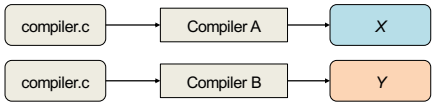
Anyone inspecting the source code to the compiler or login.c will not see any malicious code!

"No amount of source-level verification or scrutiny will protect you from using untrusted code."

Question 2

What does David A. Wheeler propose as a test to determine if a compiler has been made untrustworthy?

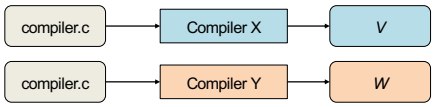
Compile the compiler source with two different compilers, producing two executable compilers, X and *Y*



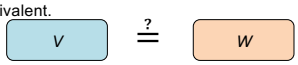
Since we're using two different compilers, *X* and *Y* will not be bit-for bit equivalent but ***X* and *Y* will be functionally equivalent**

Question 2

- Now compile the original compiler source with these two compilers, *X* and *Y*, to produce two additional executable compilers, *V* and *W*



- Since the logic of the compiler is identical and X and Y are functionally equivalent, the resulting code produced by them, V and W, should be bit-for-bit equivalent.



- If *V* and *W* are not the same then we know that one of the compilers we used initially contains the trojan horse code and inserts a backdoor.

Question 3a

NaCl is the sandbox for running native code under Chromium browsers

(a) How would an NaCl container, which is forbidden from accessing a system's storage, be able to offer store and retrieve local files if needed?

It would use the the IMC (Inter-Module Communications) mechanism to establish a communication channel to a storage service that runs as a native (trusted) browser plugin.

Question 3b

NaCl is the sandbox for running native code under Chromium browsers

(b) The inner sandbox allows us to place a trusted service runtime subsystem within the same process as the untrusted application module. What two mechanisms are used by the inner sandbox subsystem to keep applications from doing harm?

From the paper, the two mechanisms are:

1. Static analysis to detect security defects in the code and disallowed instructions through disassembly.
2. Segmented memory to constrain data and instruction references

Question 3b (continued)

The "inner sandbox uses a set of rules for reliable disassembly, a modified compilation tool chain that observes these rules, and a static analyzer that confirms that the rules have been followed."

1. Rules for disassembly – *allowable instructions & instruction sequences*
2. Modified compilation toolchain (libraries) that observes these rules
3. Static analyzer: validates that the rules are being followed

Note that the "rules" are *policy*, not *mechanism*

The end