EPFL

SPRING
SECURITY AND PRIVACY ENGINEERING LABORATORY

Computer Security (COM-301)
Malware
Introduction

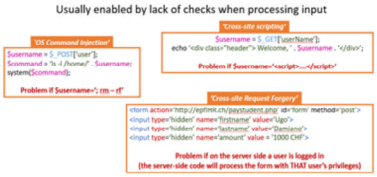
Carmela Troncoso
SPRING Lab
carmela.troncoso@epfl.ch

Some slides/ideas adapted from: Gianluca Stringhini / Emiliano de Cristofaro / George Danezis

Previous attacks: the adversary actively exploits model/ design/ implementation errors

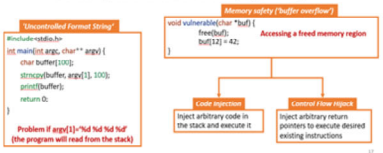
Insecure interactions between components

Usually enabled by lack of checks when processing input



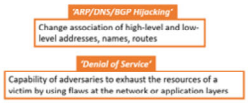
Risky Resource Management

Enabled by lack of checks or careless programming



Network attacks

Lack of security mechanisms in network protocols



Expert adversary

requires deep understanding of computer systems and networks

"Manual" adversary

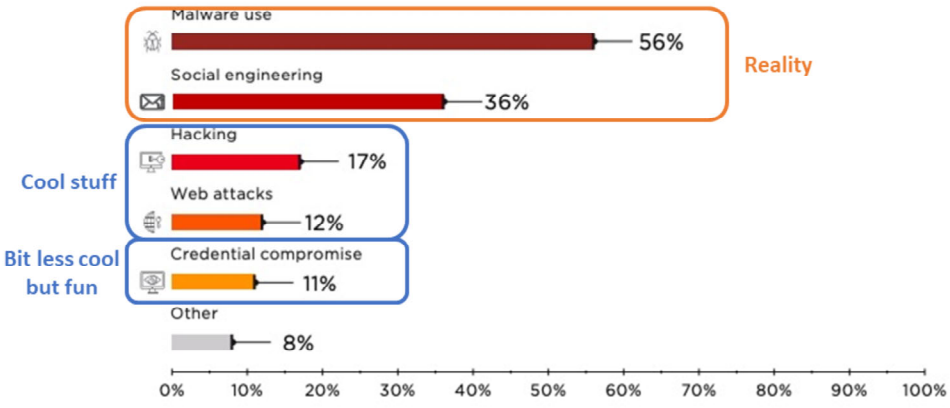
requires manual coding and testing to find the vulnerabilities and exploit them

2

n the attacks we saw in the previous lecture the adversary is an expert that understands protocols and their threat models.

For instance, the attacks on nework protocols assume threat modeling errors, namely the assumption that network nodes will be honest; or downgrade attacks on TLS exploit the fact that to make TLS compatible with previous versions the designers allow servers and clients to choose insecure primitives. On the implementation side, we saw how not sanitizing inputs allows to exploit all sorts of bugs that can result in impersonation, arbitrary commands execution, or the machine just crashing.

All these attacks have a common ground: they require the adversary to study the protocol and produce the code that exploits the vulnerability. This is very time consuming, and not all adversaries may have this knowledge.



https://www.ptsecurity.com/ww-en/analytics/cybersecurity-threatscape-2019-q1/

In fact, those attacks are not the most popular in reality. Social engineering, in which the adversary bypasses security mechanisms by getting information from users, or malware use, in which adversaries deploy malicious software to launch attacks at large scale are the most numerous.

Malware

Short for "Malicious Software"

Software that **fulfills the author's malicious intent**
Intentionally written to cause adverse effects
Many flavors with a common characteristic: **perform some unwanted activity**

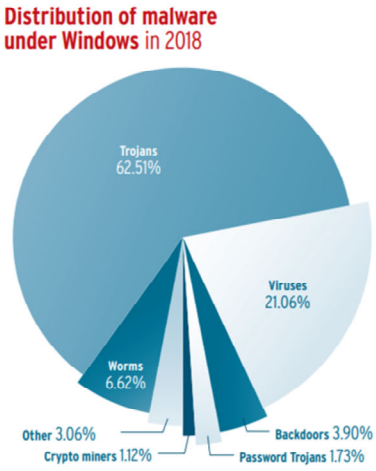
Malware != virus

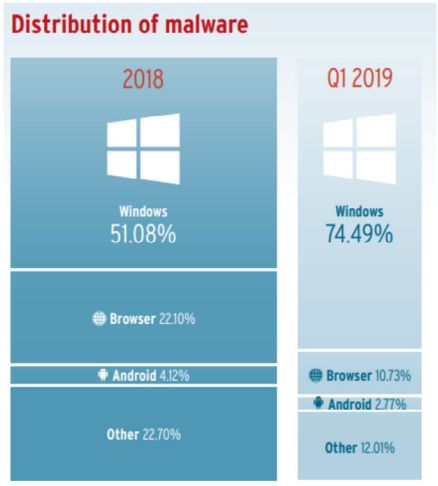
Virus is a kind of Malware

4

In this lecture we study Malware. This is software intentionally designed to do some malicious activity and create damage for hosts and users.

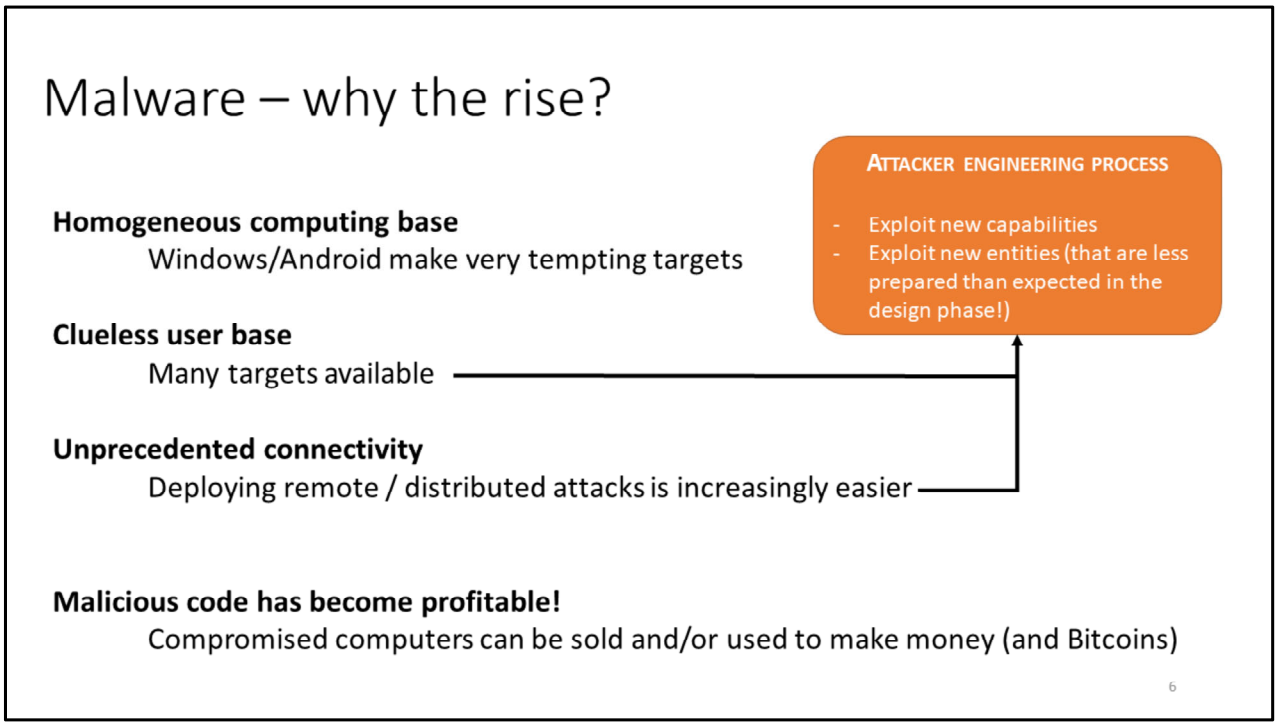
It is important to note that *virus* is only one type of malware. There are other kinds as we will see in this lecture.





https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2018-2019.pdf

In fact, Viruses are only the 21% of malware written for Windows; which is the most common target of malware, followed by browser-oriented malware and Android.



There are several reasons for the rise and dominance of Malware as the main threat for computer security.

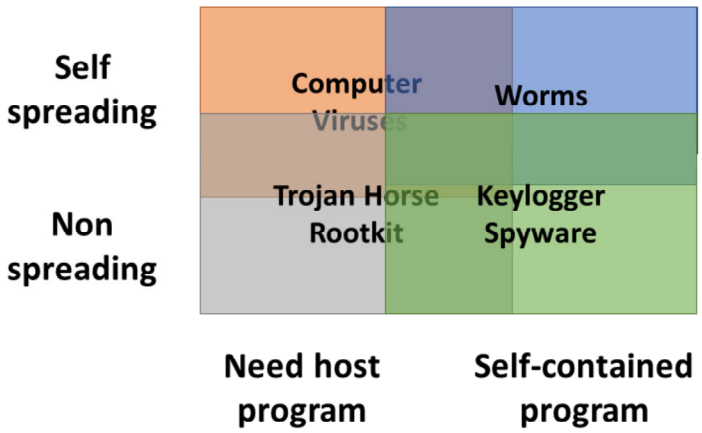
First, the increase in devices connected to the network, with some operative systems (such as Windows or Android) having a base of millions of users across the planet increase the impact of any piece of malware – one attack, millions of victims.

Computers have gone from being a work-station for experts, to be a commodity for any user. Users, thus, are not anymore experts and thus are easier targets for attacks.

Not only there are more users with less education, but their computers are connected to the network, increasing the surface of attack.

The importance on computers in everyday users' and industry activities, as well as their computational capabilities, also make compromised computers highly profitable. For instance, by requesting money from users to recover their machines, or using these machines to deploy new attacks or perform heavy computations (e.g., mininig Bitcoins).

Malware – taxonomy



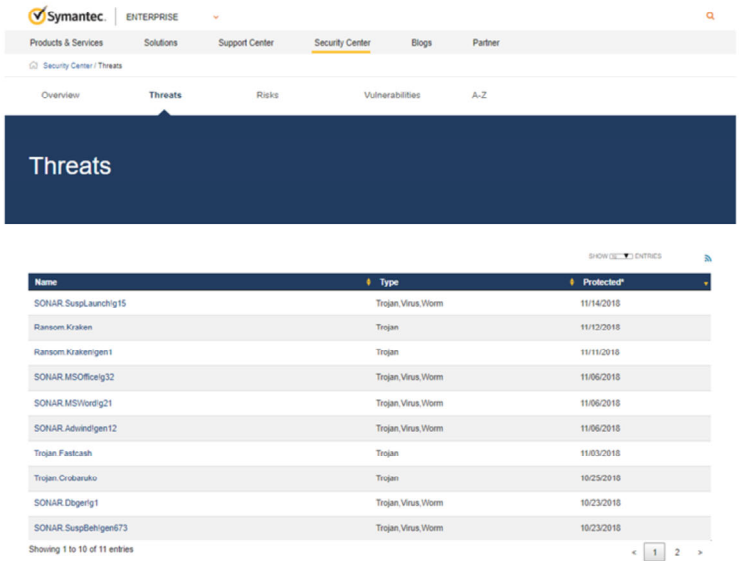
There are different types of malware. Main differences are how they spread, and whether they are self contained.

- Viruses and worms can spread by themselves. However, Viruses tend to need some human action that triggers their spread, while Worms do it on their own. Trojans or spyware do not perform any action to automate their spread and only move to new devices with deliberate downloads of compromised software.
- Viruses and Trojans typically cannot execute by themselves, they need to *infect* another program – i.e., include their code in another program and be executed with that program.

Nowadays the boundaries between these categories have become very blurry (https://www.websecurity.digicert.com/security-topics/difference-between-virus-worm-and-trojan-horse)

Malware – taxonomy

Modern malware tends to combine "the best" of the categories to achieve its purpose



[DEPRECATED]

Lists of:

- **Threats**
- **Vulnerabilities**
- **Risks**

8

Not only the boundaries have become blurry, but typically modern malware combines features from the different categories to increase its impact..

(This web from Symantec does not exist anymore after it was bought by Broadcom)

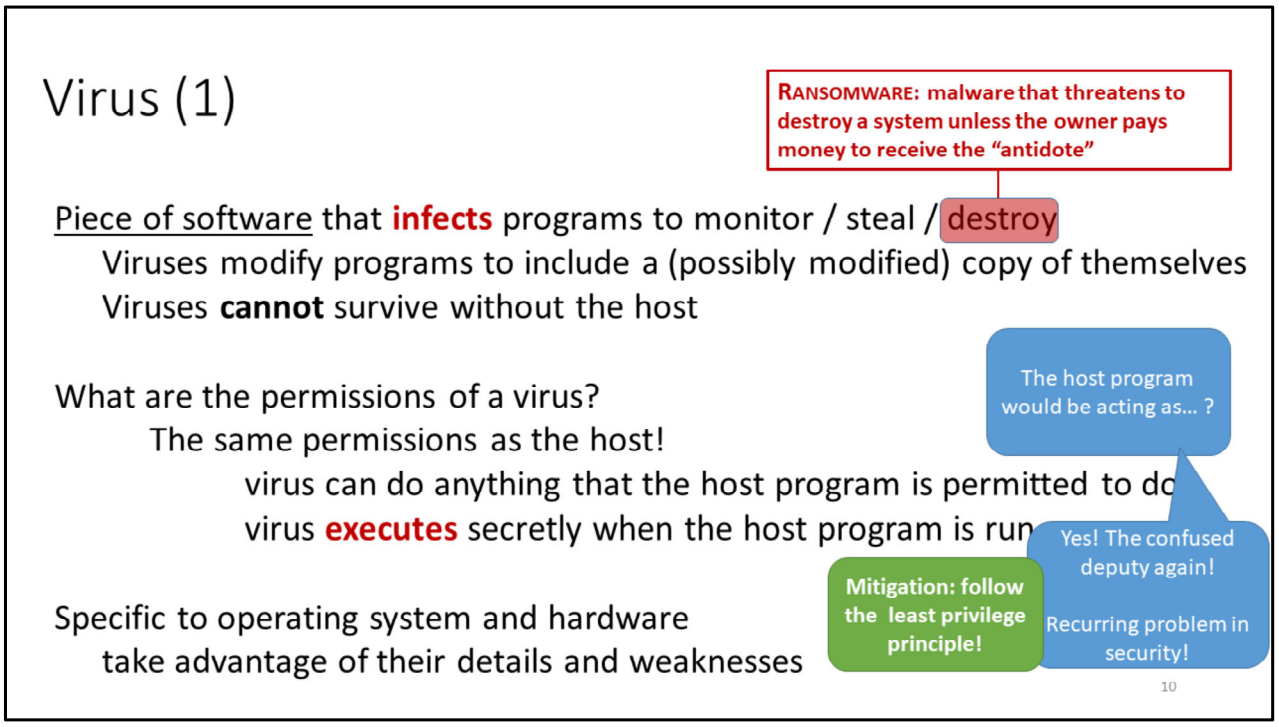
EPFL

SPRING
SECURITY AND PRIVACY ENGINEERING LABORATORY

Computer Security (COM-301)
Malware
Types of Malware

Carmela Troncoso
SPRING Lab
carmela.troncoso@epfl.ch

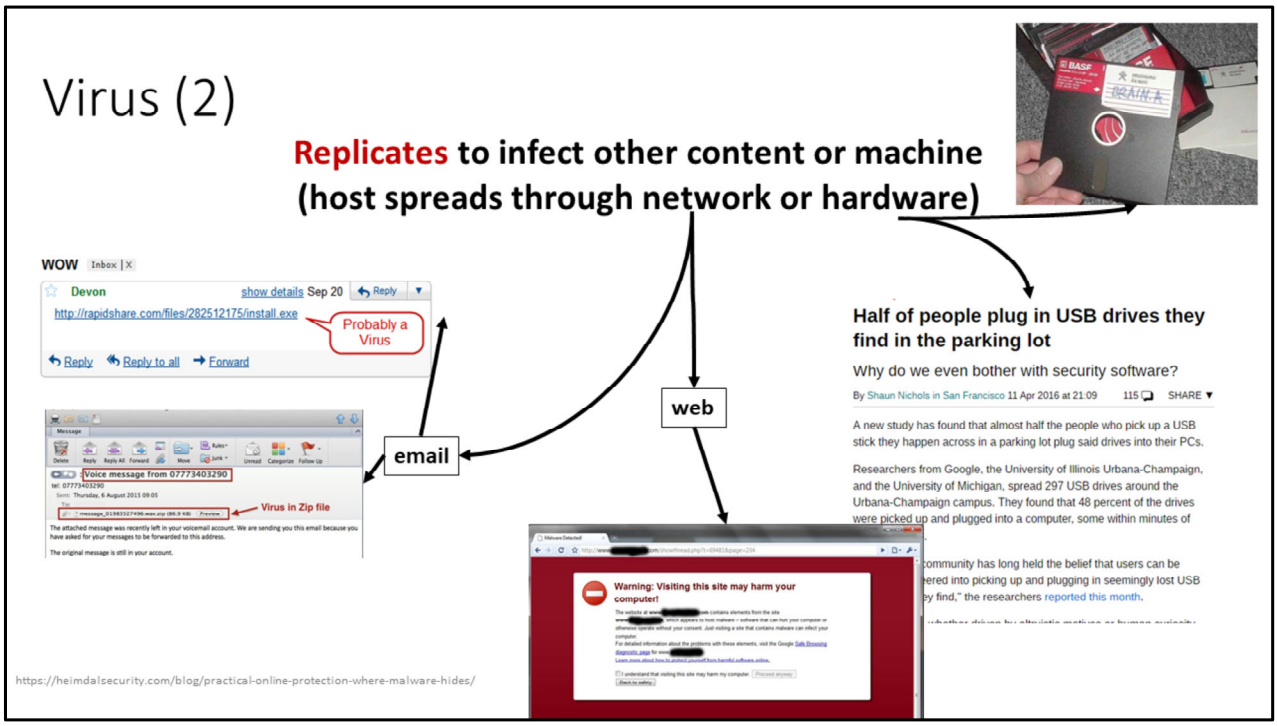
Some slides/ideas adapted from: Gianluca Stringhini / Emiliano de Cristofaro / George Danezis



A **virus** is a piece of software that *infects* (embeds itself in) other programs to perform malicious actions such as monitor users' actions, steal users' data, or destroy users' data.

The virus cannot survive or execute without the host (i.e., if the host is deleted so is the virus – though the virus may remain in the machine if it has infected more programs). When the virus executes, it does so with the permissions of the host program, which becomes a confused deputy that does malicious actions in the name of the virus. A key mitigation is to give programs the least privileges such that, even when they act in malicious ways they cannot do great damage.

Virus exploit OS-specific or hardware-specific vulnerabilities. Thus they are typically very targeted to one particular architecture and system.



Virus replicate themselves to infect contents or machines. Typically, the replication is triggered by a human action:

- Opening an email or an attachment both infects files in the host and triggers distribution
- Visiting a malicious website that downloads a virus into the computer
- Using an infected hardware support (such a usb or a diskette) that contains malicious code that infects the host as soon as it is connected.

Virus – where can they act

File infection

Overwrite (substitute the original program), *Parasitic* (append and modify)

Macro infection

Overwrite macro executed on program load (MS Excel, Word)
Need to find an exploit to insert the macro

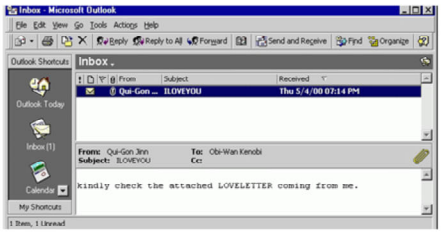
Boot infection

Most difficult! …and most dangerous
Infect booting partition

12

Viruses can infect any kind of software, from any file in the system (overwriting it or just modifying some of its pieces as part of the infection), a macro that is executed by one or more programs, or if can also infect the very operative system (e.g., the bios and the booting programs). The latter is the most dangerous, as it gets to be executed before the computer is totally started, and thus before the TCB is in place (see rootkits at the end of the lecture).

Example Virus – ILoveYou (2000)



Target: Windows 9X/2000

"LOVE-LETTER-FOR-YOU.txt.vbs" sent as email attachment

Known extension to Windows, hidden from users!
Users think they open a text file, not an executable

Operation:

Replaced files with extensions JPG, JPEG, VBS, JS, DOC, …
The script adds Windows Registry data for automatic startup on system boot
Sent itself to each entry Outlook address book, sometimes changing subject
Downloaded the Barok Trojan: "WIN-BUGSFIX.EXE" (steal passwords)

Damage: $10 billion

13

Virus – defenses

Antivirus Software

sequence of bytes/instructions that are known to be part of the virus

Signature-based detection

Database of byte-level or instruction-level signatures that match virus
Wildcards and regular expression can be used
Hash of known malicious programs

Heuristics (check for signs of infection / anomalies) and
incorrect header sizes, suspicious code section name
Behavioral signatures – detect series of changes done by a virus

Sandboxing

Run untrusted applications in restricted environment (e.g., use a VM)

14

The main defense against virus is the use of **Antivirus**. Programs that identify virus to avoid that the user executes them (avoiding infection, malicious activity, or further spread) and to indicate which files have to be cleaned or removed from the system.

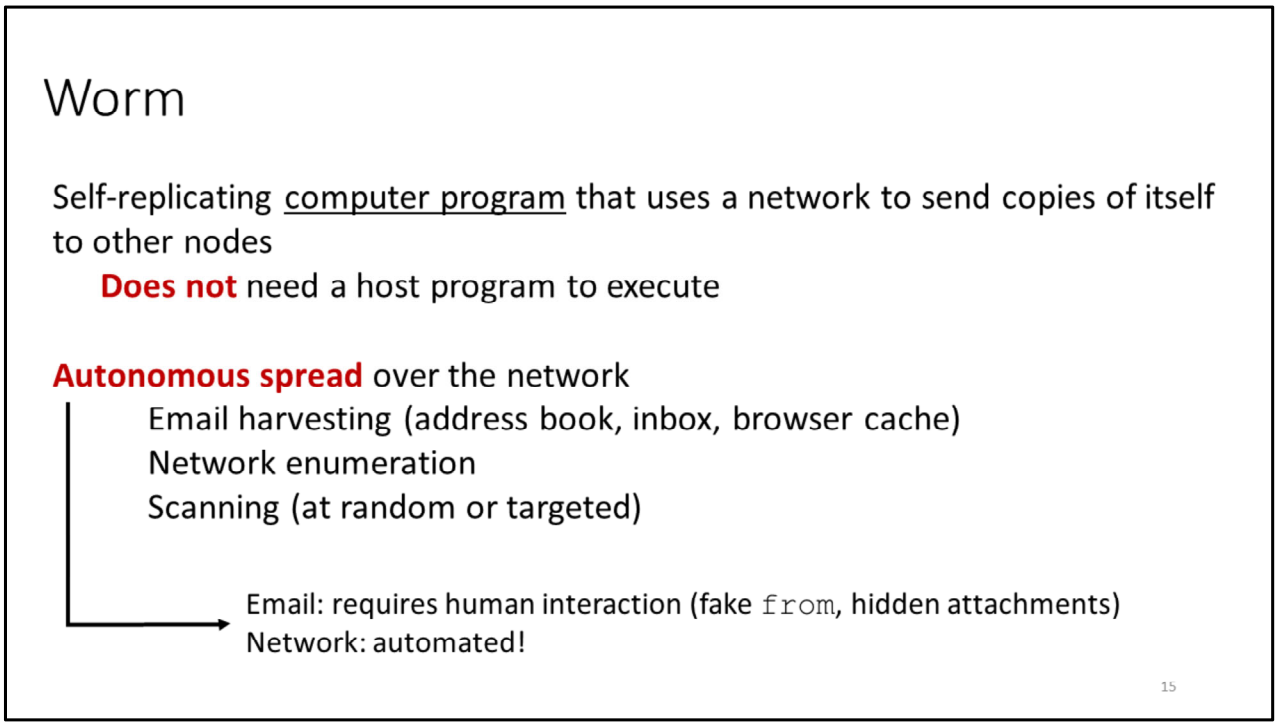
Antivirus typically have two ways of identifying virus:

- *Signature-based:* in this mode, antivirus try to find exact *signatures* in the host. Signatures are pieces of code (at byte or instruction levels) that match previously identified virus. Signatures can also be made of regular expressions, including wildcards, or can be full programs (e.g., by comparing a hash of the executable).

- *Heuristic-based:* in this mode, antivirus try to find patterns that are known to be produced by viruses, e.g., series of access to the registry, systematic changes in function headers, etc.

Signature-based produce very few false positives (i.e., false alarms), but can only find versions of virus for which their signature is known. Heuristics-based can be more flexible and also identify virus mutations, at the cost of increased false positives.

Besides using an antivirus, the effect of the virus can be reduced by running untrusted applications with least privileges (i.e., **sandboxing** the application).



A **worm** is a standalone (i.e., *does not need a host to be executed*) piece of software that can generate malicious actions.

A worm can spread autonomously, i.e., does not necessarily need a human action to replicate itself (although replication can also be triggered use human actions, e.g., by opening an email attachment).

To decide where to replicate the worm can find addresses at the application layer – such as emails in the infected host (e.g., address book, emails in inbox, etc) -- or at the network level – such as scanning the network to learn which IPs are reachable, or directly enumerate all possible address and try to infect them if any machine is found at those addresses.

Worm example 1: Code Red (2001)

Exploits known buffer overflow in Microsoft IIS (web server)

Propagates through an HTTP request (TCP)

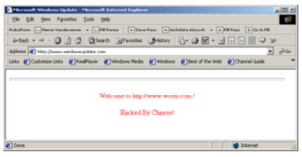
Achieve the overflow

```
GET /default.ida?NNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNN
%u9090%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u7801
%u9090%u6858%ucbd3%u7801%u9090%u9090%u8190%u00c3
%u0003%u8b00%u531b%u53ff%u0078%u0000%u00=a  HTTP/1.0
```

Payload: injected executable code

Lack of Sanitization!!

Operation



1st Defacing:

2nd Date dependent:

Day 1-19: continue spreading (random IPs)

Day 20-27: Launch Denial of Service (among them White House)

Day 28-end: Sleep

Vulnerable population (360,000 servers) infected in 14 hours

16

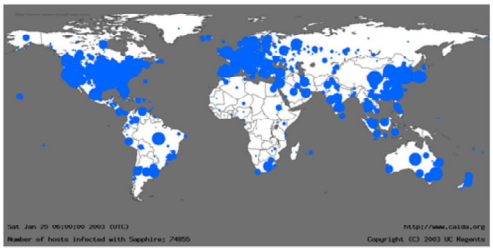
Worm example 2: Slammer (2003)

Fastest worm ever

Exploits a vulnerability (buffer overflow) in Microsoft SQL Server

Creates random IPs and sends itself (small 380 byte UDP packet)

75,000 victims infected within ten minutes
Doubled infections every 8.5 seconds



Consequences – Internet denied of service:
saturated many Internet links
routers collapsed
affected ATMs and emergency numbers
root DNS shut down

17

Worm example 2: Slammer (2003)

Fastest worm ever

Exploits a vulnerability (buffer overflow) in M

Creates random IPs and sends itself (small 38

75,000 victims infected within ten minutes
Doubled infections every 8.5 seconds

Consequences – Internet denied of service:
saturated many Internet links
routers collapsed
affected ATMs and emergency numbe
root DNS shut down

Security

Slammer worm slithers back online to attack ancient SQL servers

If you get taken down by this 13-year-old malware, you probably deserve it

By Darren Pauli 5 Feb 2017 at 23:29 11 SHARE ▼

One of the world's most famous net menaces, SQL Slammer, has resumed attacking servers some 13 years after it set records by infecting 75,000 servers in 10 minutes, researchers say.

The in-memory worm exploits an ancient flaw in Microsoft SQL server and Desktop Engine triggering denial of service, and at the time of its emergence significantly choking internet traffic.

Researcher Michael Bacarella first raised the alarm to Slammer which was created on the back of public proof-of-concept exploit code published during Black Hat by now Google security boffin David Litchfield.

Check Point researchers detected re-emergent attacks in early December, noting that most targeted machines in the US.

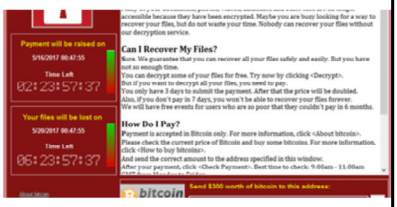
"More than a decade later, Slammer is hitting again," researchers say.

18

Worm example - WannaCry (2017)

A case of **Ransomware**

→ **Require money to recover system**



Exploits a vulnerability revealed in a NSA hacking toolkit leak

- Mishandled packets for the Microsoft Server Message Block (protocol for shared access) enable arbitrary code execution
- The leak contained vulnerabilities in systems from e.g., Cisco Systems and Fortinet Inc

Encrypted data and asked for ransom in Bitcoins

- 300$ in 3 days or 600$ in 7 days or **DELETE**

>200,000 victims

$130,634 obtained in ransom

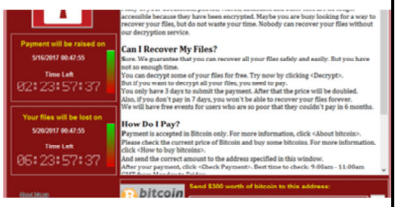
billions of dollars in damage, UK Hospitals affected

19

Worm example - WannaCry (2017)

A case of **Ransomware**

→ Require money to recover system



How did it end?

The worm "kill switch" was found

Upon installation, the malware checked the existence of a web. If yes, it stopped.

A researcher registered the website and the worm stopped

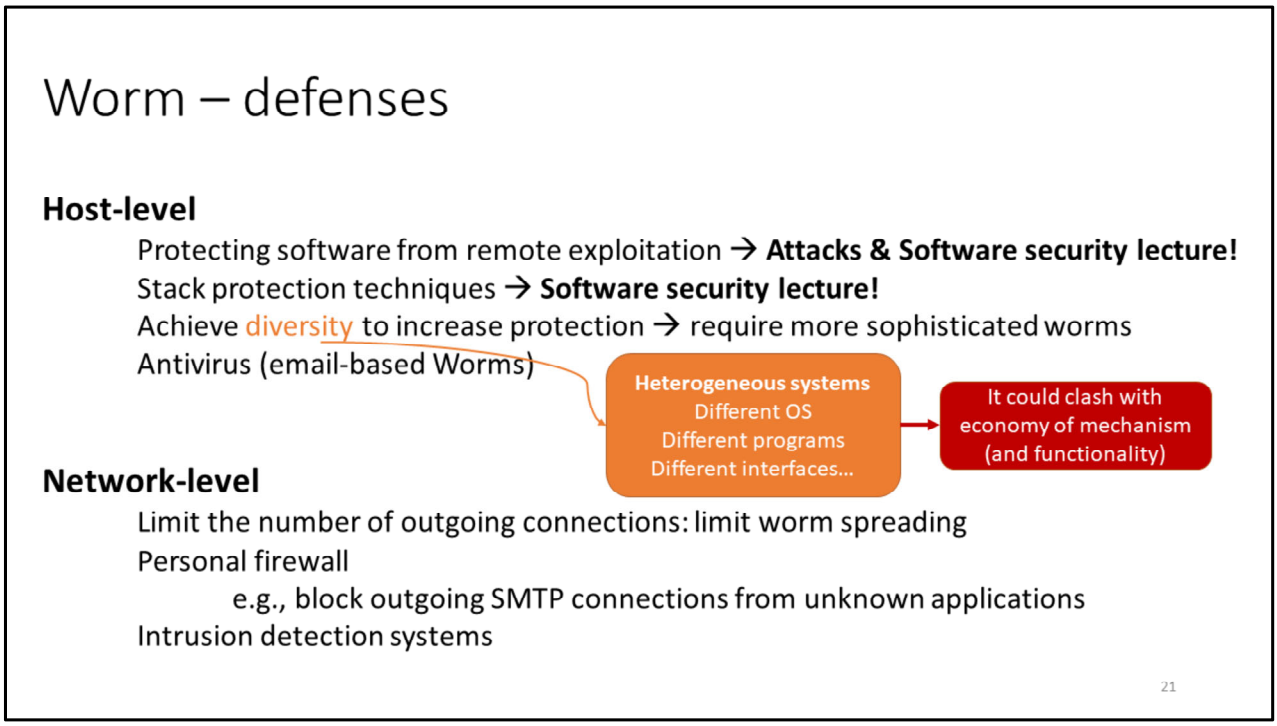
Why have a kill switch?

Avoid worm study if hijacked, or if in sandbox

20

Some worms have a "kill switch" that makes them stop their spread. Kill switches are used for hackers both to stop the spread of the worm once monetization has been successful, but also to avoid that sexurity researchers can analyze the worm in controlled conditions (e.g., by running it in a sandbox).

For the particular case of WannaCry, the kill switch was the existence of a website. When the worm arrived to a new host, the first thing it did was to check the existence of a web by launching a DNS request on a domain hardcoded in the worm code. A researcher studying the worm saw this domain and registered it. This effectively stopped the worm.



Host-level protection

First, Worms infect machines by exploiting vulnerabilities on hosts. Protections such as those seen in the software security and attacks lectures that reduce the possibility of exploitation will mitigate the impact of the worm.

Second, Like virus, worms may have signatures or recognizable behaviours that can be identified by an antivirus.

Finally, diversity in OS/programs/interfaces/ help avoiding wide compromise of a systems. See this as a separation of privilege: the worm needs to be able to find (and exploit) vulnerabilities in all this environments.

Network-level protection

A way to mitigate the damage done by worms is to limit their capability to spread. This can be done by limiting the amount of outgoing connections from a host or a network. Another option is to have rules that avoid network connections that are inconsistent with typical behavior (e.g., sending emails from applications that are not a mail client).

Finally, one can also try to detect Intrusions, i.e., detecting the worm when it is trying to infect the system.

Intrusion detection systems – what they do

Host-based vs. Network-based

Host: process running on a host. Detects local malware

Network: network appliance monitoring all traffic

Signature based vs. Anomaly-based detection

Signature: identifies known patterns

+ low false alarms

- expensive (need up-to-date signatures), can't find new attacks

Anomaly: attempts to identify behavior different than legitimate

+ adapt to new attacks (legitimate does not change!)

- high number of false alarms

22

An **Intrusion Detection System (IDS)** aims at identifying when the system is under an attack from a malicious entity. IDS can run on a *host* aiming at detecting malware attacking the host; or at the *network level* inspecting traffic aiming at identifying malware attacking hosts in the network via patterns in the traffic.

IDSs can be classified in a similar way as antivirus:

- **Signature-based:** where the IDS tries to find known patterns in connections or host processes. As in the antivirus case, this produces very few false alarms at the cost of only being able to detect already known attacks.

- **Anomaly-based:** where the IDS tries to model what normal behavior is, and to identify any deviation and tag it as malicious. This approach holds great potential to detect new attacks: no matter what the new attack does, the legitimate traffic does not change and does the attack is still identifiable. On the downside this approach is prone to produce a high number of false alarms as legitimate programs may many times do actions that are different from their normal behavior.

Note: Intrusion Detection Systems are not limited to detecting worms and can be used

to identify other types of malicious behavior.

Trojan Horse

Malware that *appears to perform a desirable function* but it also performs **undisclosed malicious activities**

- Requires users to **explicitly** run the program

→ **Defense: Train users!**

and follow least privilege principle!

Cannot replicate but can do any malicious activity

- Spy on sensitive user data (spyware)
- Allow remote access (backdoor)
- Base for further attacks→ act as mail relay (for spammers)
- Damage routines (corrupting files)

23

A **Trojan horse** is a piece of software that performs (or appears to perform) a desirable function (e.g., cleaning the hard drive, improve the performance of the machine), but at the same time is performing another malicious function in the background.

Trojans cannot run on their own. They require that the user executes the program that contains the Trojan. As such, good defenses are running untrusted programs with least privilege, and also train the user to not run programs that come from untrusted sources and therefore may contain malicious code.

Trojans **cannot replicate by themselves.** They require users to download or transfer the program with the Trojan.

Trojan Horse examples:
Tiny Banker Trojan (2012)
Gameover Zeus (2013)

Goal: steal users sensitive data, such as account login information and banking codes.

Mode of Operation 1

1. Sniff packets to learn when a user visits **a banking website**
2. Steal credentials before they are sent → send to malware server
   Reads keystrokes before encryption!!

Mode of Operation 2

1. Sniff packets to learn when a user visits **a banking website**
2. Steal appearance from website
3. Ask questions to user on a pop-up → send answer to malware server

24

Rootkit

Adversary controlled code that takes residence **deep within the TCB** of a system
- Hides his presence by modifying the OS

Installed by an attacker **after** a system has been compromised
- Difficult to detect
  - Replace system programs with trojaned versions
  - Modify kernel data structures to hide processes, files, and network activities
- Allows the adversary to return on a later time

Defense (difficult!): Integrity checkers user/kernel level

25

A **Rootkit** is malicious code that the adversary has managed to install in the core of the Operative System, and thus inside the Trusted Computing Base. To get to this point, the adversary typically has to first compromise the system to be able to run inside and then installs this code. Once the code is installed it can replace system's programs with Trojans that perform malicious activities.

Rootkits are extremely dangerous because they run inside the TCB. Thus, *there is no protection in place!* (recall that everything in the TCB is trusted by definition and does not need to be checked). As they are in the TCB defending is hard. One has to look for anomalies that hint that something weird is happening within the system.

Furthermore, rootkits are extremely difficult to detect. As they are inside the OS and can act on booting, they can hide themselves by modifying the OS and kernel structures such that their actions are invisible.

Rootkits also typically open backdoors to let the adversary come back to the user.

Rootkit+Worm example: Stuxnet (2010)

Goal: Attack SCADA (Control systems) of Iran's nuclear power plants

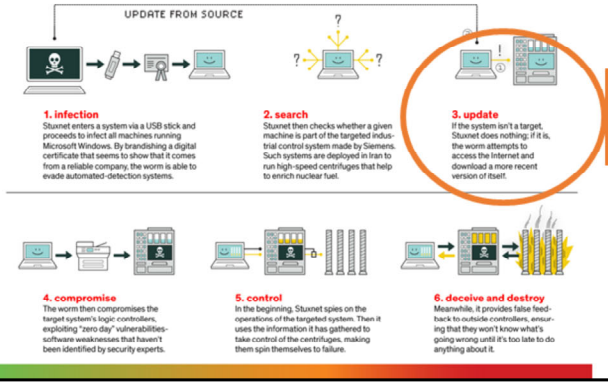
Three modules:

Worm: spread Stuxnet's payload

Link file: executed malicious code

Rootkit: hide the presence of malicious file to avoid detection

Stuxnet needs to be in the network, but the network is closed and disconnected
→ Entered via infected USB



Very targeted attack. If the system infected is not a target, the malware stays dormant.

Authorship?

Alleged Israel/US Cyberweapon

https://spectrum.ieee.org/telecom/security/the-real-story-of-stuxnet/

Backdoor

A **hidden** functionality that allows the adversary to bypass some security mechanism

Why not "audit" the program?

We can audit the program source

what if the **compiler** is malicious and introduces backdoors?

Chain of reasoning leads us to suspect all programs down to the very first compiler!

Key paper: *Thompson, Ken. "Reflections on trusting trust." Communications of the ACM (1984)*

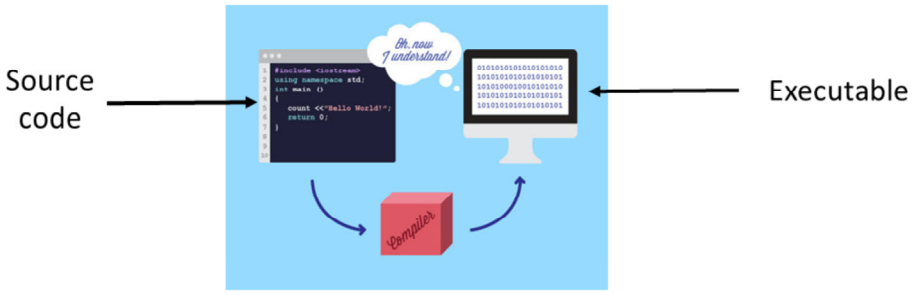
More readable summary: https://www.schneier.com/blog/archives/2006/01/countering_trus.html 27

A **Backdoor** is a hidden functionality that enables the user to skip some security mechanism (e.g., opens ports that are not protected by the policy, open connections to applications without authentication, etc.)

An idea to find out whether a program has a backdoor (or a Trojan, or whether an OS has a rootkit installed) is to audit the program. A first step can be to inspect the source code, but even if the source code is clean, a backdoor can be introduced by the compiler. One could then think to audit the compiler, but the same problem arises… at the end of the day one has to trust the first compiler.

Cheatsheet: what is a compiler?

Computer software that transforms high-level code (basically any programming language) into low-level code that can be understood by the machine



Example compilers: gcc (C language), javac (Java language)

https://s3.amazonaws.com/kukuruku-co/uploads/images/59/03/5903d81e484b2.jpg

https://dwheeler.com/trusting-trust/

Backdoor

How can we avoid blind trust on compilers? (How can we avoid trusting trust?)

Challenge: you have the executable of two compilers C1 (Exe_{C1}) and C2 (Exe_{C2}). You want to know if they are hiding a backdoor.

1. Start with another compiler C_A source and compile it with the two compiler executables Exe_{C1} and Exe_{C2}

   Compile C_A with the first compiler: $Exe_{C1}(C_A) = Exe_{A,1}$

   Compile C_A with the second compiler: $Exe_{C2}(C_A) = Exe_{A,2}$

29

A trick to check for the existence of backdoors is to follow the separation of privilege principle.

Suppose we have two compilers C1 and C2 for which we suspect they may introduce backdoors on the programs it compiles.

And another compiler C_A source. One can compile the compiler C_A with the two suspect executables. The result should be two versions of the same compiler, i.e., when they are applied to another program they should result on the same executable.

https://dwheeler.com/trusting-trust/

Backdoor

How can we avoid blind trust on compilers? (How can we avoid trusting trust?)

Challenge: you have the executable of two compilers C1 ($ExeC_1$) and C2 ($ExeC_2$). You want to know if they are hiding a backdoor.

2. Use the compiler executables $Exe_{A,1}$ and $Exe_{A,2}$ to compile C_A again

   We expect that: $\mathbf{Exe_{A,1}(C_A)=Exe_{A,2}(C_A)}$ **Why?**

   $Exe_{A,1}$ and $Exe_{A,2}$ are executables of the same compiler C_A**!**
   Therefore given the same input they should output the same executable code

3. If not, one of them (or both) are introducing a backdoor!

30

Then, we can use the two executable versions of C_A to compile C_A again. As both executables are from the same compiler, the result should be the same.

If this is not the case, and there are some differences one can conclude that one compiler (or both compilers) are introducing a backdoor that changes the execution of the programs they compile, In this case they change the execution of the compiler C_A.

https://dwheeler.com/trusting-trust/

Backdoor

What principle have we used here?

Separation of privilege

Have two compilers → the adversary needs to subvert both in the same way!

How can [illegible] rusting trust?)

Challeng[illegible] and C2 ($ExeC_2$). You want to know if they are hiding a backdoor.

1. Start with another compiler C_A source and compile it with the two compiler executables Exe_{C1} and Exe_{C2}

   $Exe_{C1}(C_A) = Exe_{A,1}$ $\quad\quad$ $Exe_{C2}(C_A) = Exe_{A,2}$

2. Use the compiler executables $Exe_{A,1}$ and $Exe_{A,2}$ to compile C_A again

   The two binaries should now be the same! **$Exec_{A,1}(C_A)=Exec_{A,2}(C_A)$?**

3. If not, one of them (or both) are introducing a backdoor!

31

Computer Security (COM-301)
Malware
Botnets

Carmela Troncoso
SPRING Lab
carmela.troncoso@epfl.ch

Some slides/ideas adapted from: Gianluca Stringhini / Emiliano de Cristofaro / George Danezis

Botnets

Attacks at scale!!

Multiple (millions) compromised **hosts** under the control of a single entity

"zombies" or "bots"

uses

Bot-net command & control (C&C)

System to keep track of bots and send commands to them

33

Up to here we have seen attacks that can be automatized by malware, but are still launched from one machine, or from uncoordinated machines.

Botnets are groups of compromised hosts (up to millions) under the control of one entity that coordinates their actions. Compromised hosts are called **bots** or **zombies** (as many times they are computers connected to the internet that do not interact, and are in zombie state until the entity controlling them launches an attack). The single entity that controls the bots uses a **botnet Command & Control center (C&C)**. This is a system, composed by one or more machines, that enables the entity to send commands to the bots.

Botnets

Attacks at scale!!

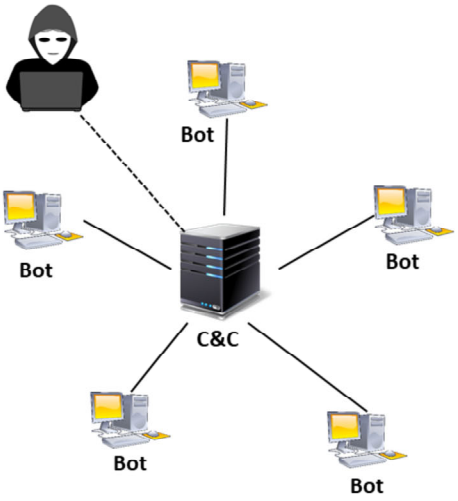
Multiple (millions) compromised **hosts** under the control of a single entity

34

Up to here we have seen attacks that can be automatized by malware, but are still launched from one machine, or from uncoordinated machines.

Botnets are groups of compromised hosts (up to millions) under the control of one entity that coordinates their actions. Compromised hosts are called **bots** or **zombies** (as many times they are computers connected to the internet that do not interact, and are in zombie state until the entity controlling them launches an attack). The single entity that controls the bots uses a **botnet Command & Control center (C&C)**. This is a system, composed by one or more machines, that enables the entity to send commands to the bots.

Botnets - Star Topology



What is the problem here?

C&C single point of failure

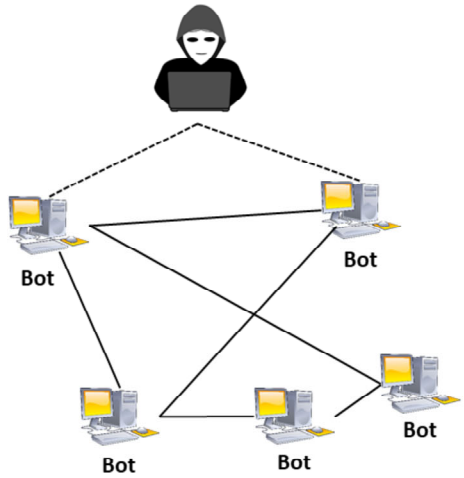
the botnet violates the *least common mechanism* principle!

35

The simplest option to centralize the Command & Control (C&C) on one machine controlled by the hacker. This is simple to manage, but also implies that the C&C machine is a *single point of failure.* If the machine is taken down, the hacker loses control of the botnet.

Also, one machine may not scale if the number of bots in the botnet is larger than thousands or millions of machines.

Botnets – P2P Topology



No Command and Control!!

Difficult management (join? leave?)

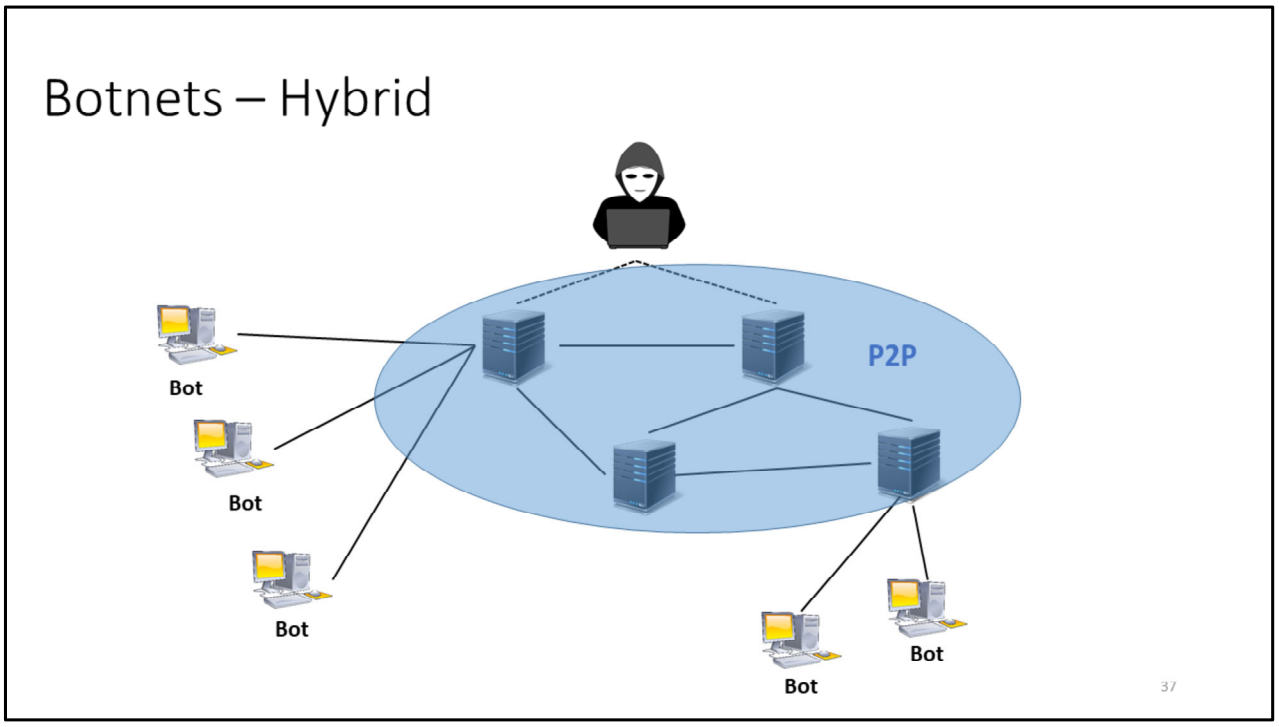
Vulnerable to attacks in which too many bots are taken over (these are called Sybil attacks)

36

On the opposite side to a centralized C&C design, is a totally decentralized option in which there is no machine that issues commands, but the bots themselves issue commands. The hacker would communicate with some of the bots and these would pass the commands to the other bots in a peer-to-peer (P2P) fashion.

This scheme has high resilience. Even if some of the bots are taken down, the network can still operate. However, it is very difficult to manage: how can a node enter or leave the network? How do bots know with whom they have to communicate?

It also opens the door to *Sybil attacks* in which security defenders gain control of enough bots in the network to counter the malicious commands issued by the hacker.



In general, botnets follow an approach in the middle. Few nodes form the C&C talking to each other in a P2P fashion, under the strict control of the hacker. As there are few, there are easy to manage (e.g., they could be enumerated by the hacker that can tell each of them who are their neighbours in the network).

Each of the C&C nodes can take care of issuing commands to a subset of the bots in the network. Even though in the figure bots are only connected to one C&C node, in reality it may be desirable to connect them to more than one for resilience: i.e., if one C&C node is taken down, the bots are still reachable.

Monetizing Botnets

Rental – "Pay me money, and I'll let you use my botnet…"

DDoS extortion – "Pay me or I take down you legitimate business"

Bulk traffic selling – "Pay me to boost visit counts on your website"

Click fraud – "Simulate clicks on advertised links to generate revenue"

Distribute Ransomware – "I've encrypted your hard drive, pay!"

Advertise products – "Pay me, I will leave comments all around the web"

Bitcoin mining!!

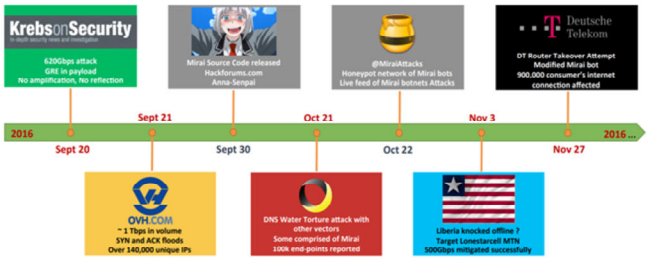
…

Botnets can be used for many purposes. These purposes are typically profitable for the botnet owner.

Example Botnet – Mirai (2016)

Target: IoT devices

scanning of Telnet ports, attempted to log in using 61 username/password combos



Open source code – variants appear all the time

Wicked (2018): scans ports 8080, 8443, 80, and 81 and attempts to locate vulnerable, unpatched IoT devices running on those ports.

https://www.nanog.org/sites/default/files/1_Winward_Mirai_The_Rise_v1.pdf
https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-antonakakis.pdf

39

Botnets: defense

Attack C&C infrastructure

Take communication channel off-line
Hijack/poison DNS to route traffic to black hole

Honeypots

Vulnerable computer that serves no purpose other than to attract attackers and study their behavior in controlled environments

Study botnet behavior to find defense (or study ecosystem)

40

In order to take down a botnet, one must take down the C&C infrastructure, either by taking the nodes down or isolating them so that they cannot communicate with the nodes. The latter can be done by taking the communication offline, or by rerouting the traffic from/to bots to a black hole, e.g., by hijacking or poisoning the DNS domain of the C&C.

To take down the C&C one needs to find it. A useful tool to discover where the C&C is is the deployment of honeypots – machines that are vulnerable on purpose so that the botnet takes over them and then their behavior can be studied.

Other malware

Rabbit: code that replicates itself w/o limit to exhaust resources

Logic (time) bomb: code that triggers action when condition (time) occurs

Dropper: code that drops other malicious code

Tool/toolkit: program used to assemble malicious code (not malicious itself)

Scareware: false warning of malicious code attack

Summary

Malware = software intentionally malicious

Can be exploited by non-experienced adversaries

Many types depending on
(auto) replication, need for a host

Botnets – attacks at scale!