anti-virus and anti-anti-virus
logistics: TRICKY

HW assignment out

“infecting” an executable
anti-virus techniques

last time: signature-based detection
  regular expression-like matching
  snippets of virus(-like) code

heuristic detection
  look for “suspicious” things

behavior-based detection
  look for virus activity

not explicitly mentioned: producing signatures
  manual? analysis

not explicitly mentioned: “disinfection”
  manual? analysis
anti-virus techniques

last time: signature-based detection
  regular expression-like matching
  snippets of virus(-like) code

heuristic detection
  look for “suspicious” things

behavior-based detection
  look for virus activity

not explicitly mentioned: producing signatures
  manual? analysis

not explicitly mentioned: “disinfection”
  manual? analysis
regular expression cheatsheet

a — matches a

a* — matches (empty string), a, aa, aaa, ...

a\ — matches the string a*

foo|bar — matches foo, bar

[ab] — matches a, b

[^ab] — matches any byte except a and b

( foo|bar )* — (empty string), foo, bar, foobar, barfoo, ...

(.|\n)* — matches anything whatsoever
recall: why regular expressions?

(essentially) one-pass, lookup table

not the most flexible, but fast

flex — regular expressions + code for exceptions
recall: faster than regular expressions?

optimization 1: look for fixed-length strings
   sliding window + hashtable
   test with full pattern

optimization 2: head/tail scanning
   avoid reading whole files
scanning for fixed strings

16-byte “anchor” | malware
-----------------|-------------------
204D616C696369667573205468696E6720 | Virus A
34567890ABCDEF023456789ABCDEF0345 | Virus B
6120766972757320737472696E679090F2 | Virus C
... | ...
scanning for fixed strings

| 123456789ABCDEF0 | 23456789ABCDEF034567... |

hash function

<table>
<thead>
<tr>
<th>4-byte hash</th>
<th>malware</th>
</tr>
</thead>
<tbody>
<tr>
<td>204D616C0FC923131</td>
<td>96E6720 Virus A</td>
</tr>
<tr>
<td>3456780A34598873</td>
<td>EFG0345 Virus B</td>
</tr>
<tr>
<td>612076697994254A3</td>
<td>79090F2 Virus C</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
scanning for fixed strings

<table>
<thead>
<tr>
<th>4-byte hash</th>
<th>malware</th>
</tr>
</thead>
<tbody>
<tr>
<td>204D616C0FC923131</td>
<td>96E6720</td>
</tr>
<tr>
<td>34567890A34598873</td>
<td>EFG0345</td>
</tr>
<tr>
<td>612076697994254A3</td>
<td>79090F2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

(full pattern for Virus B)
scanning for fixed strings

<table>
<thead>
<tr>
<th>4-byte hash</th>
<th>malware</th>
</tr>
</thead>
<tbody>
<tr>
<td>FC923131</td>
<td>Virus A</td>
</tr>
<tr>
<td>34598873</td>
<td>Virus B</td>
</tr>
<tr>
<td>994254A3</td>
<td>Virus C</td>
</tr>
</tbody>
</table>

(full pattern for Virus B)
scanning for fixed strings

<table>
<thead>
<tr>
<th>byte sequence</th>
<th>4-byte hash</th>
<th>malware</th>
</tr>
</thead>
<tbody>
<tr>
<td>204D616C6F</td>
<td>FC923131</td>
<td>Virus A</td>
</tr>
<tr>
<td>34567890</td>
<td>994254A3</td>
<td>Virus B</td>
</tr>
<tr>
<td>61207669</td>
<td>79090F2</td>
<td>Virus C</td>
</tr>
</tbody>
</table>

(full pattern for Virus B)
virus patterns

specific — large snippet of code from virus
   false positives essentially impossible

general — strategy (e.g. push + ret)
   false positives possible
   real applications might do this?
   might appear in application data?
detecting new malware

goal: detect unseen malware

some signatures might do this — look for strategies

also look for anomalies

hope that real compilers/linkers/etc. don’t do …
anti-virus techniques

last time: signature-based detection
  regular expression-like matching
  snippets of virus(-like) code

heuristic detection
  look for “suspicious” things

behavior-based detection
  look for virus activity

not explicitly mentioned: producing signatures
  manual? analysis

not explicitly mentioned: “disinfection”
  manual? analysis
### Viruses and Executable Formats

<table>
<thead>
<tr>
<th><strong>Header</strong></th>
<th>Machine type, file type, etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Program Header</strong></td>
<td>“Segments” to load (also, some other information)</td>
</tr>
<tr>
<td><strong>Segment 1 Data</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Segment 2 Data</strong></td>
<td></td>
</tr>
</tbody>
</table>

- **Heuristic 1**: Is entry point in last segment? (Last segment usually not code)
- **Heuristic 2**: Did virus mess up header? (e.g. do sizes used by linker but not loader disagree? Section names disagree with usage?)
## Viruses and Executable Formats

- **Header**: Machine type, file type, etc.
- **Program header**: "segments" to load (also, some other information)
  - Length edited by virus
- **Segment 1 data**
- **Segment 2 data**
  - Virus code + new entry point?

### Heuristics
1. **Is entry point in last segment?**
   - Last segment usually not code
2. **Did virus mess up header?**
   - (e.g., do sizes used by linker but not loader disagree)
   - Section names disagree with usage?
viruses and executable formats

<table>
<thead>
<tr>
<th>header: machine type, file type, etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>program header: “segments” to load</td>
</tr>
<tr>
<td>(also, some other information)</td>
</tr>
<tr>
<td>heuristic 1: is entry point in last segment?</td>
</tr>
<tr>
<td>(last segment usually not code)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>segment 1 data</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>segment 2 data</td>
</tr>
<tr>
<td>virus code + new entry point?</td>
</tr>
<tr>
<td>header: machine type, file type, etc.</td>
</tr>
<tr>
<td>--------------------------------------</td>
</tr>
<tr>
<td>program header: “segments” to load</td>
</tr>
<tr>
<td>(also, some other information)</td>
</tr>
<tr>
<td>new segment added by virus</td>
</tr>
<tr>
<td>segment 1 data</td>
</tr>
<tr>
<td>segment 2 data</td>
</tr>
<tr>
<td>segment 3 data — virus segment</td>
</tr>
</tbody>
</table>
viruses and executable formats

<table>
<thead>
<tr>
<th>header</th>
<th>machine type, file type, etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>program header</td>
<td>“segments” to load (also some other info)</td>
</tr>
<tr>
<td>heuristic 1</td>
<td>is entry point in last segment? (last segment usually not code)</td>
</tr>
<tr>
<td>segment 1 data</td>
<td></td>
</tr>
<tr>
<td>segment 2 data</td>
<td></td>
</tr>
<tr>
<td>segment 3 data — virus segment</td>
<td></td>
</tr>
</tbody>
</table>
## Viruses and Executable Formats

<table>
<thead>
<tr>
<th><strong>Header</strong></th>
<th>Machine type, file type, etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Program Header</strong></td>
<td>“Segments” to load</td>
</tr>
<tr>
<td>Heuristic 2</td>
<td>Did virus mess up header?</td>
</tr>
<tr>
<td></td>
<td>(e.g. do sizes used by linker but not loader disagree)</td>
</tr>
<tr>
<td></td>
<td>Section names disagree with usage?</td>
</tr>
<tr>
<td><strong>Segment 2 Data</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Segment 3 Data</strong></td>
<td>Virus segment</td>
</tr>
</tbody>
</table>
defeating entry point checking

insert jump in normal code section and...
   set as entry point; or
   assume it’s reached ‘soon’
defeating entry point checking

insert jump in normal code section and...

set as entry point; or
assume it’s reached ‘soon’

“dynamic” heuristic: run code in VM for a while, see if switches sections
heuristics: library calls

dynamic linking — functions called by name

how do viruses add to dynamic linking tables?
often don't! — instead dynamically look-up functions
if do — could mess that up/lots of code

heuristic: look for API function name strings
(outside linking info)
evading library call checking

modify dynamic linking tables

reimplement library call manually
  Linux: usually easy
  Windows: system calls not well documented, change

hide names
evading library call checking

modify dynamic linking tables

reimplement library call manually
  Linux: usually easy
  Windows: system calls not well documented, change

hide names
hiding library call names

common approach: store hash of name

runtime: read library, scan list of functions for name

bonus: makes analysis harder
anti-virus techniques

last time: signature-based detection
   regular expression-like matching
   snippets of virus(-like) code

heuristic detection
   look for “suspicious” things

behavior-based detection
   look for virus activity

not explicitly mentioned: producing signatures
   manual? analysis

not explicitly mentioned: “disinfection”
   manual? analysis
behavior-based detection

things malware does that other programs don’t?
behavior-based detection

things malware does that other programs don’t?

modify system files

modifying existing executables

open network connections to lots of random places

...
behavior-based detection

things malware does that other programs don’t?

modify system files

modifying existing executables

open network connections to lots of random places

... 

monitor all programs for weird behavior

problem: false positives (e.g. installers)
heuristic detection

virus “shortcuts”

- generally: not producing executable via normal linker
- generally: trying to make analysis harder
- push then ret instead of jmp
- entry point in “wrong” segment
- switching segments
- library calls without normal dynamic linker mechanisms

infection behavior

- modifying executables/system files
- weird network connections
example heuristics: DREBIN (1)


features from applications (without running):
  - hardware requirements
  - requested permissions
  - whether it runs in background, with pushed notifications, etc.
  - what API calls it uses
  - network addresses

detect dynamic code generation explicitly

statistics (i.e. machine learning) to determine score
example heuristics: DREBIN (2)

advantage: Android uses Dalvik bytecode (Java-like)
    high-level “machine code”
    much easier/more useful to analyze

accuracy?
    tested on 131k apps, 94% of malware, 1% false positives
    versus best commercial: 96%, < 0.3% false positives
        (probably has explicit patterns for many known malware samples)

...but
    statistics: training set needs to be typical of malware
    cat-and-mouse: what would attackers do in response?
anti-virus techniques

last time: signature-based detection
   regular expression-like matching
   snippets of virus(-like) code

heuristic detection
   look for “suspicious” things

behavior-based detection
   look for virus activity

not explicitly mentioned: producing signatures
   manual? analysis

not explicitly mentioned: “disinfection”
   manual? analysis
anti-anti-virus

defeating signatures:

avoid things compilers/linkers never do

make analysis harder
  - takes longer to produce signatures
  - takes longer to produce “repair” program

make changing viruses
  - make any one signature less effective
some terms

armored viruses
viruses designed to make analysis harder

metamorphic/polymorphic/oligomorphic viruses
viruses that change their code each time
different terms — different types of changes (later)
encrypted(?) data

```c
char obviousString[] =
    "Please_open_this_100%"
    "safe_attachment";
char lessObviousString[] =
    "oSZ^LZ\037POZQ\037KWVL\037\016\017"
    \017\032\037L^YZ\037^KK^\WRZQK";
for (int i = 0; i < sizeof(lessObviousString) - 1
    lessObviousString[i] =
        lessObviousString[i] ^ '?';
}
recall: hiding API calls

/* functions, functionsNames retrieved from library before */
/* 0xd7c9e758 = hash("GetFileAttributesA") */
unsigned hashOfString = 0xd7c9e758;
for (int i = 0; i < num_functions; ++i) {
    unsigned functionHash = 0;
    for (int j = 0; j < strlen(functionNames[i]); ++j) {
        functionHash = (functionHash * 7 + functionNames[i][j]);
    }
    if (functionHash == hashOfString) {
        return functions[i];
    }
}
encrypted data and signatures

doesn’t really stop signatures
   “encrypted” string + decryption code is more unique

but makes analyzing virus a little harder
   how much harder?
exercise: how would you decrypt strings?
encrypted data and signatures

doesn’t really stop signatures
   “encrypted” string + decryption code is more unique

but makes analyzing virus a little harder
   how much harder?
   exercise: how would you decrypt strings?

can we do better?
encrypted viruses

char encrypted[] = "\x12\x45...";
char key[] = "...";

virusEntryPoint() {
    decrypt(encrypted, key);
    goto encrypted;
}

decrypt(char *buffer, char *key) {...}

choose a new key each time!

not good encryption — key is there

sometimes mixed with compression
encrypted viruses: no signature?

decrypt is a pretty good signature

still need to a way to disguise that code

how about analysis? how does one analyze this?
not just anti-antivirus

“encrypted” body

just running objdump not enough...

instead — run debugger, set breakpoint after “decryption”

dump decrypted memory afterwards
unneeded steps

understanding the “encryption” algorithm
   more complex encryption algorithm won’t help

extracting the key and encrypted data
   making key less obvious won’t help
unneeded steps

understanding the “encryption” algorithm
  more complex encryption algorithm won’t help

extracting the key and encrypted data
  making key less obvious won’t help

needed to know when encryption finished

needed debugger to work
unneeded steps

understanding the “encryption” algorithm
more complex encryption algorithm won’t help

extracting the key and encrypted data
making key less obvious won’t help

needed to know when encryption finished

needed debugger to work

countermeasures?
encrypt in strange order? multiple passes?
anti-debugging (later)
example: Cascade decrypter

```
lea encrypted_code, %si
decrypt:
  mov $0x682, %sp  // length of body
  xor %si, (%si)
  xor %sp, (%si)
  inc %si
  dec %sp
  jnz decrypt
encrypted_code:
  ...
```
example: Cascade decrypter

```assembly
lea encrypted_code, %si

decrypt:
    mov $0x682, %sp  // length of body
    xor %si, (%si)
xor %sp, (%si)
    inc %si
    dec %sp
    jnz decrypt

encrypted_code:
    ...
```

Szor Listing 7.1
example: Cascade decrypter

```
ela encrypted_code, %si
decrypt:
  mov $0x682, %sp  // length of body
  xor %si, (%si)
  xor %sp, (%si)
  inc %si
  dec %sp
  jnz decrypt
encrypted_code:
  ...
```
decrypter

more variations:
   nested decrypters, different orders, etc.

still problem: decrypter code is signature

...but harder to distinguish different malware

often tries to frustrate debugging in other ways
   e.g. use stack pointer (not for the stack)
   (more on this later)
decrypter

more variations: nested decrypters, different orders, etc.

still problem: decrypter code is signature

...but harder to distinguish different malware

often tries to frustrate debugging in other ways e.g. use stack pointer (not for the stack)

“disinfection” — want to precisely identify malware
decrypter

more variations:
  nested decrypters, different orders, etc.

still problem: decrypter code is signature

...but harder to distinguish different malware

often tries to frustrate debugging in other ways

  e.g. (more)
  easiest way to defeat decrypter manually: run in debugger until code is decrypted
legitimate “packers”

some commercial software is packaged in this way

...including antidebugging stuff

why? intended to be copy/reverse engineering protection
playing mouse

signature-based techniques:
   scan for pattern of constant part of virus
   scan for strings, approx. 16-bytes long
   shortcut: scan top and bottom

virus-writer hat: how can you defeat these?
   encrypting code? — encrypter is pattern
playing mouse

signature-based techniques:

- scan for pattern of constant part of virus
- scan for strings, approx. 16-bytes long
- shortcut: scan top and bottom

virus-writer hat: how can you defeat these?

encrypting code? — encrypter is pattern

change some trivial part of virus — e.g. add nops somewhere
adding nops

instead of copying, copy but insert nops

a little tricky — only between instructions

could have hard-coded places to insert
  likely easy to turn into signature
  or tricky to write

or can parse instructions
  x86 encoding isn’t that bad
  malware can use limited subset
producing changing malware

not just \texttt{nop}:

switch between synonym instructions

swap registers

random instructions that manipulate ‘unused’ register

...
oligomorphic viruses

use packing technique but

make slight changes to decrypters
### example: W95/Memorial

```assembly
mov $0x405000, %ebp
mov $0x550, %ecx
lea 0x2e(%ebp), %esi
add 0x29(%ebp), %ecx
mov 0x2d(%ebp), %al
```

```assembly
decrypt:
  nop
  nop
  xor %al, (%esi)
  inc %esi
  nop
  inc %al
  dec %ecx
  jnz decrypt
  ...
```

```assembly
mov $0x550, %ecx
mov $0x13bc000, %ebp
lea 0x2e(%ebp), %esi
add 0x29(%ebp), %ecx
mov 0x2d(%ebp), %al
```

```assembly
decrypt:
  nop
  nop
  xor %al, (%esi)
  inc %esi
  nop
  inc %al
  loop decrypt
  ...
```

--

Szor, Listings 7.3 and 7.4
example: W95/Memorial

```assembly
mov $0x405000, %ebp
mov $0x550, %ecx
lea 0x2e(%ebp), %esi
add 0x29(%ebp), %ecx
mov 0x2d(%ebp), %al

decrypt:
nop
nop
xor %al, (%esi)
inc %esi
nop
inc %al
dec %ecx
jnz decrypt
...
```

change instruction order; location of decryption key/etc.

```assembly
mov $0x550, %ecx
mov $0x13bc000, %ebp
lea 0x2e(%ebp), %esi
add 0x29(%ebp), %ecx
mov 0x2d(%ebp), %al

decrypt:
nop
nop
xor %al, (%esi)
inc %esi
nop
inc %al
loop decrypt
...
```

Szor, Listings 7.3 and 7.4
example: W95/Memorial

```assembly
mov $0x405000, %ebp
mov $0x550, %ecx
lea 0x2e(%ebp), %esi
add 0x29(%ebp), %ecx
mov 0x2d(%ebp), %al

decrypt:
  nop
  nop
  xor %al, (%esi)
  inc %esi
  nop
  inc %al
  dec %ecx
  jnz decrypt
  ...

mov $0x550, %ecx
mov $0x13bc000, %ebp
lea 0x2e(%ebp), %esi
add 0x29(%ebp), %ecx
mov 0x2d(%ebp), %al

nop
nop
xor %al, (%esi)
inc %esi
nop
inc %al
loop decrypt
...`
```

Szor, Listings 7.3 and 7.4
example: W95/Memorial

mov $0x405000, %ebp
mov $0x550, %ecx
lea 0x2e(%ebp), %esi
add 0x29(%ebp), %ecx
mov 0x2d(%ebp), %al

mov $0x550, %ecx
mov $0x13bc000, %ebp
lea 0x2e(%ebp), %esi
add 0x29(%ebp), %ecx
mov 0x2d(%ebp), %al

decrypt:
    nop
    nop
    xor %al, (%esi)
    inc %esi
    nop
    inc %al
dec %ecx
jnz decrypt

Szor: “96 different decryptor patterns”

...
more advanced changes?

Szor calls W95/Memorial oligomorphic
   “encrypted” code
   plus small changes to decrypter

What about doing more changes to decrypter?
   many, many variations

Szor calls doing this polymorphic

polymorphic example: 1260
example: 1260 (virus)

```assembly
inc %si
mov $0x0e9b, %ax
clc
mov $0x12a, %di
nop
mov $0x571, %cx
decrypt:
xor %cx, (%di)
sub %dx, %bx
sub %cx, %bx
sub %ax, %bx
nop
xor %cx, %dx
xor %ax, (%di)
...
```  

```assembly
mov $0x0a43, %ax
nop
mov $0x15a, %di
sub %dx, %bx
sub %cx, %bx
mov $0x571, %cx
clc
decrypt:
xor %cx, (%di)
xor %cx, %dx
sub %cx, %bx
nop
xor %cx, %bx
xor %ax, (%di)
...
```

adapted from Szor, Listing 7.5
example: 1260 (virus)

```assembly
inc %si
mov $0x0e9b, %ax
clc
mov $0x12a, %di
nop
mov $0x571, %cx

decrypt:
xor %cx, (%di)
sub %dx, %bx
sub %cx, %bx
sub %ax, %bx
nop
xor %cx, %dx
xor %ax, (%di)
...
```

```assembly
mov $0x0a43, %ax
nop
mov $0x15a, %di
sub %dx, %bx
sub %cx, %bx
mov $0x571, %cx
clc

decrypt:
xor %cx, (%di)
xor %cx, %dx
xor %cx, %bx
nop
xor %cx, %bx
xor %ax, (%di)
...
```

adapted from Szor, Listing 7.5
example: 1260 (virus)

inc %si
mov $0x0e9b, %ax
clc
mov $0x12a, %di
nop
mov $0x571, %cx

decrypt:
oxor %cx, (%di)
sub %dx, %bx
sub %cx, %bx
sub %ax, %bx
nop
xor %cx, %dx
xor %ax, (%di)
...

mov $0x0a43, %ax
nop
mov $0x15a, %di
sub %dx, %bx
sub %cx, %bx
mov $0x571, %cx

decrypt:
oxor %cx, (%di)
oxor %cx, %dx
xor %cx, %dx
xor %ax, (%di)
...

adapted from Szor, Listing 7.5
example: 1260 (virus)

```assembly
inc %si
mov $0x0e9b, %ax
clc
mov $0x12a, %di
nop
mov $0x571, %cx

deencrypt:
xor %cx, (%di)
sub %dx, %bx
sub %cx, %bx
sub %ax, %bx
nop
xor %cx, %dx
xor %ax, (%di)
...
```

```assembly
mov $0x0a43, %ax
nop
mov $0x15a, %di
sub %dx, %bx
sub %cx, %bx
mov $0x571, %cx
clc
deencrypt:
xor %cx, (%di)
xor %cx, %dx
sub %cx, %bx
nop
xor %cx, %bx
xor %ax, (%di)
...
```

adapted from Szor, Listing 7.5
example: 1260 (virus)

```assembly
inc %si
mov $0x0e9b, %ax
clc
mov $0x12a, %di
nop
mov $0x571, %cx
decrypt:
xor %cx, (%di)
sub %dx, %bx
sub %cx, %bx
sub %ax, %bx
nop
mov $0x0a43, %ax
nop
mov $0x15a, %di
sub %dx, %bx
sub %cx, %bx
mov $0x571, %cx
decrypt:
xor %cx, (%di)
xor %cx, %dx
sub %cx, %bx
nop
xor %cx, %bx
xor %ax, (%di)
...  
adapted from Szor, Listing 7.5
```

different decryption “key”
lots of variation

essentially **limitless variations** of decrypter

  huge number of *nop*-like sequences
  plus reordering *non-nop* instructions

can’t just make scanner that skips obvious *nops*
lots of variation

essentially **limitless variations** of decrypter
  huge number of **nop**-like sequences
  plus reordering non-**nop** instructions

can’t just make scanner that skips obvious **nops**

could try to analyze more deeply for **nops**
  could identify when instruction’s result is unused
lots of variation

essentially **limitless variations** of decrypter

  huge number of **nop**-like sequences
  plus reordering non-nop instructions

can’t just make scanner that skips obvious nops

could try to analyze more deeply for nops
  could identify when instruction’s result is unused

but attacker can be more sophisticated:

```
inc %ax; dec %ax
xor %ax, %bx; xor %bx, %ax; xor %ax, %bx
...```

43
interlude: anti-packer strategies
finding packers

easiest way to decrypt self-decrypting code — run it!
solution: virtual machine in antivirus software

makes antivirtualization/emulation more important
finding packers with VM

run program in VM for a while
   how long?

then scan memory for known patterns

or detect jumping to written memory
stopping packers

it’s unusual to jump to code you wrote

modern OSs: memory is executable or writable — not both
stopping packers

it’s unusual to jump to code you wrote

modern OSs: memory is executable or writable — not both
diversion: $\text{DEP}/W^X$

memory executable or writeable — but not both

exists for exploits (later in course), not packers

requires hardware support to be fast (early 2000s+)

various names for this feature:

Data Execution Prevention (DEP) (Windows)
$W^X$ (“write XOR execute”)
NX/XD/XN bit (underlying hardware support)
(No Execute/eXecute Disable/eXecute Never)

special system call to switch modes
unusual, but...

binary translation
  convert machine code to new machine code at runtime

Java virtual machine, JavaScript implementations
  “just-in-time” compilers

dynamic linkers
  load new code from a file — same as writing code?

those packed commercial programs

programs need to explicitly ask for write+exec
finding packers

easiest way to decrypt self-decrypting code — run it!
solution: virtual machine in antivirus software

makes antivirtualization/emulation more important
antivirtualization techniques

query virtual devices

time operations that are slower in VM/emulation

use operations not supported by VM
antivirtualization techniques

query virtual devices

time operations that are slower in VM/emulation

use operations not supported by VM
virtual devices

VirtualBox device drivers?

VMware-brand ethernet device?

...
antivirtualization techniques

query virtual devices
  solution: mirror devices of some real machine

time operations that are slower in VM/emulation

use operations not supported by VM
antivirtualization techniques

query virtual devices

time operations that are slower in VM/emulation

use operations not supported by VM
slower operations

not-“native” VM:
  everything is really slow

otherwise — trigger “callbacks” to VM implementation:
  system calls?
  allocating and accessing memory?

...and hope it’s reliably slow enough
antivirtualization techniques

query virtual devices

time operations that are slower in VM/emulation
  solution: virtual clock

use operations not supported by VM
antivirtualization techniques

query virtual devices

time operations that are slower in VM/emulation

use operations not supported by VM
operations not supported

missing instructions kinds?
  FPU instructions
  MMX/SSE instructions
  undocumented (!) CPU instructions

not handling OS features?
  setting up special handlers for segfault
  multithreading
  system calls that make callbacks
  ...

antivirus not running system VM to do decryption
  needs to emulate lots of the OS itself
attacking emulation patience

looking for unpacked virus in VM

...or other malicious activity

when are you done looking?
attacking emulation patience

looking for unpacked virus in VM

...or other malicious activity

when are you done looking?

malware solution: take too long
    not hard if emulator uses “slow” implementation
attacking emulation patience

looking for unpacked virus in VM

...or other malicious activity

when are you done looking?

malware solution: take too long

    not hard if emulator uses “slow” implementation

malware solution: don’t infect consistently
if (randomNumber() == 4) {
    unpackAndRunEvilCode();
}

antivirus emulator:
randomNumber() == 3
looks clean!

real execution #1:
randomNumber() == 2
no infection!

real execution #N:
randomNumber() == 4
infect!
on goats

analysis (and maybe detection) uses goat files

“sacrificial goat” to get changed by malware

heuristics can avoid simple goat files, e.g.:
  don’t infect small programs
  don’t infect huge programs
  don’t infect programs with huge amounts of nops
  ...

60
goats as detection

tripwire for malware
touching do-nothing.exe — very likely bad
goats as analysis

more important for analysis of changing malware
want examples of multiple versions
want it to be obvious where malware code added
  e.g. big cavities to fill in original
  e.g. obvious patterns in original code/data
changing bodies

“decrypting” a virus body gives body for “signature”
“just” need to run decrypter

how about avoiding static signatures entirely

called metamorphic
versus polymorphic — only change “decrypter”
example: changing bodies

pop %edx
mov $0x4h, %edi
mov %ebp, %esi
mov $0xC, %eax
add $0x88, %edx
mov (%edx), %ebx
mov %ebx, 0x1118(%esi,%eax,4)

pop %eax
mov $0x4h, %ebx
mov %ebp, %esi
mov $0xC, %edi
add $0x88, %eax
mov (%eax), %esi
mov %esi, 0x1118(%esi,

code above: after decryption

every instruction changes

still has good signatures
    with alternatives for each possible register selection

but harder to write/slower to match
case study: Evol


“mutation engine”
run as part of propagating the virus

code

\[
\text{disassemble} \rightarrow \text{instr. lengths} \rightarrow \text{transform} \rightarrow \text{relocate}
\]
case study: Evol


“mutation engine”

run as part of propagating the virus

code

\[ \text{disassemble} \rightarrow \text{instr. lengths} \rightarrow \text{transform} \rightarrow \text{relocate} \]
Evol instruction lengths

sounds really complicated?

virus only handles instructions it has:
  about 61 opcodes, 32 of them identified by first four bits
    e.g. opcode 0x7x – conditional jump

no prefixes, no floating point

only %reg or $constant or offset(%reg)

“mutation engine”
run as part of propagating the virus

code

\[ \text{disassemble} \rightarrow \text{instr. lengths} \rightarrow \text{transform} \rightarrow \text{relocate} \]
Evol transformations

some stuff left alone

static or random one of $N$ transformations

example:

```plaintext
push %ecx
mov %ebp, %ecx
add $0x12, %ecx
mov %eax, −0xa(%ecx)
pop %ecx
```

uses more stack space — save temporary code gets bigger each time

Lakhotia et al., “Are metamorphic viruses really invincible?”, Virus Bulletin, Jan 2005
case study: Evol


“mutation engine”
run as part of propagating the virus

code

```
<table>
<thead>
<tr>
<th>disassemble</th>
<th>instr. lengths</th>
<th>transform</th>
<th>relocate</th>
</tr>
</thead>
</table>
```

code
mutation with relocation

table mapping old to new locations
  list of number of bytes generated by each transformation

list of locations references in original
  record relative offset in jump
  record absolute offset in original
relocation example

`mov ...`

`mov ...`

`decrypt:`

`xor %rax, (%rbx)`

`inc %rbx`

`dec %rcx`

`jne decrypt`

<table>
<thead>
<tr>
<th>orig. len</th>
<th>new len</th>
<th>instr</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>10</td>
<td>mov1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>mov2</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>xor1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>incl1</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>dec1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>jne1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>address loc</th>
<th>orig. target</th>
<th>new target</th>
</tr>
</thead>
<tbody>
<tr>
<td>10+3+7+1+5+1 (jne1+1)</td>
<td>xor1 (5+2)</td>
<td>xor1 (10+3)</td>
</tr>
</tbody>
</table>
mutation engines

tools for writing polymorphic viruses

best: no constant bytes, no “no-op” instructions

tedious work to build state-machine-based detector
  ((almost) a regular expression to match it)
  apparently done manually
  automatable?

pattern: used until reliably detected
fancier mutation

can do mutation on *generic* machine code

“just” need full disassembler

identify both *instruction lengths* and *addresses*

hope machine code not written to rely on machine code sizes, etc.

hope to identify *tables of function pointers*, etc.
**fancier mutation**

also an infection technique
  no “cavity” needed — create one

obviously tricky to implement
  need to fix all executable headers
  what if you misparse assembly?
  what if you miss a function pointer?

example: Simile virus
antiantivirus

already covered:
  break disassemblers — with packers
  break VMs/emulators

break debuggers
  make analysis harder

break antivirus software itself
  “retrovirus”
antiantivirus

already covered:
  break disassemblers — with packers
  break VMs/emulators

break debuggers
  make analysis harder

break antivirus software itself
  “retrovirus”
**diversion: debuggers**

we’ll care about two pieces of functionality:

**breakpoints**
- debugger gets control when certain code is reached

**single-step**
- debugger gets control after a single instruction runs
we’ll care about two pieces of functionality:

breakpoints
    debugger gets control when certain code is reached

single-step
    debugger gets control after a single instruction runs
implementing breakpoints

idea: change

```
movq %rax, %rdx
addq %rbx, %rdx  // BREAKPOINT HERE
subq 0(%rsp), %r8
...
```

into

```
movq %rax, %rdx
jmp debugger_code
subq 0(%rsp), %r8
...
```
implementing breakpoints

idea: change

```plaintext
movq %rax, %rdx
addq %rbx, %rdx // BREAKPOINT HERE
subq 0(%rsp), %r8
...
```

into

```plaintext
movq %rax, %rdx
jmp debugger_code
subq 0(%rsp), %r8
...
```

problem: jmp might be bigger than addq?
x86 breakpoint instruction: `int 3`
   Why 3? fourth entry in table of handlers

`one byte` instruction encoding: CC

debugger modifies code to insert breakpoint
   has copy of original somewhere

invokes handler setup by OS
   debugger can ask OS to be run by handler
   or changes pointer to handler directly on old OSes
int 3 handler

kind of exception handler

   recall: exception handler = way for CPU to run OS code

x86 CPU saves registers, PC for debugger

x86 CPU has easy to way to resume debugged code from handler
detecting int 3 directly (1)

checksum running code

mycode:

...  
  movq $0, %rbx  
  movq $mycode, %rax

loop:
  addq (%rax), %rbx  
  addq $8, %rax
  cmpq $endcode, %rax
  jl loop
  cmpq %rbx, $EXPECTED_VALUE
  jne debugger_found
...

endcode:
detecting int 3 directly (2)

query the “handler” for int 3
   old OSs only; today: cannot set directly

modern OSs: ask if there’s a debugger attached

...or try to attach as debugger yourself
   doesn’t work — debugger present, probably
   does work — broke any debugger?

// Windows API function!
if (IsDebuggerPresent()) {

int 3 is the oldest x86 debugging mechanism

modern x86: 4 “breakpoint” registers (DR0–DR3) contain address of program instructions need more than 4? sorry

processor triggers exception when address reached 4 extra registers + comparators in CPU?

flag to invoke debugger if debugging registers used enables nested debugging
diversion: debuggers

we’ll care about two pieces of functionality:

breakpoints
  debugger gets control when certain code is reached

single-step
  debugger gets control after a single instruction runs
implementing single-stepping (1)

set a breakpoint on the following instruction?

```assembly
movq %rax, %rdx
addq %rbx, %rdx  // ←— STOPPED HERE
subq 0(%rsp), %r8  // ←— SINGLE STEP TO HERE
subq 8(%rsp), %r8
...
```

transformed to

```assembly
movq %rax, %rdx
addq %rbx, %rdx  // ←— STOPPED HERE
int 3  // ←— SINGLE STEP TO HERE
subq 8(%rsp), %
...
```

then jmp to addq
implementing single-stepping (1)

set a breakpoint on the following instruction?

```
movq %rax, %rdx
addq %rbx, %rdx  // ←− STOPPED HERE
subq 0(%rsp), %r8  // ←− SINGLE STEP TO HERE
subq 8(%rsp), %r8
...
```

transformed to

```
movq %rax, %rdx
addq %rbx, %rdx  // ←− STOPPED HERE
int 3  // ←− SINGLE STEP TO HERE
subq 8(%rsp), %
...
```

then jmp to addq
implementing single-stepping (2)

typically **hardware support** for single stepping

x86: `int 1` handler (second entry in table)

x86: TF flag: execute handler after every instruction

...except during handler (whew!)
Defeating single-stepping

try to install your own int 1 handler
  (if OS allows)

try to clear TF?
  (if debugger doesn’t reset it)
unstealthy debuggers

is a debugger installed?

unlikely on Windows, maybe ignore those machines

is a debugger process running (don’t check if it’s tracing you)

...
confusing debuggers

“broken” executable formats

- e.g., recall ELF: segments and sections
- corrupt sections — program still works
- overlapping segments/sections — program still works

use the stack pointer not for the stack
stack trace?
antiantivirus

already covered:
  break disassemblers — with packers
  break VMs/emulators

break debuggers
  make analysis harder

break antivirus software itself
  “retrovirus”
attacking antivirus (1)

how does antivirus software scan new things?
register handlers with OS/applications — new files, etc.

how about registering your own?
hooking

hooking — getting a ‘hook’ to run on (OS) operations
  e.g. creating new files

ideal mechanism: OS support

less ideal mechanism: change library loading
  e.g. replace ‘open’, ‘fopen’, etc. in libraries

less ideal mechanism: replace OS exception (system call) handlers
  very OS version dependent
hooking — getting a ‘hook’ to run on (OS) operations
  e.g. creating new files

ideal mechanism: OS support

less ideal mechanism: change library loading
  e.g. replace ‘open’, ‘fopen’, etc. in libraries

less ideal mechanism: replace OS exception (system call) handlers
  very OS version dependent
What Is a File System Filter Driver?

Last Updated: 1/24/2017

A file system filter driver is an optional driver that adds value to or modifies the behavior of a file system. A file system filter driver is a kernel-mode component that runs as part of the Windows executive.

A file system filter driver can filter I/O operations for one or more file systems or file system volumes. Depending on the nature of the driver, filter can mean log, observe, modify, or even prevent. Typical applications for file system filter drivers include antivirus utilities, encryption programs, and hierarchical storage management systems.
hooking — getting a ‘hook’ to run on (OS) operations

- e.g. creating new files

ideal mechanism: OS support

less ideal mechanism: change library loading
- e.g. replace ‘open’, ‘fopen’, etc. in libraries

less ideal mechanism: replace OS exception (system call) handlers
- very OS version dependent
changing library loading

e.g. install new library — or edit loader, but ...

not everything uses library functions

what if your wrapper doesn’t work exactly the same?
hooking — getting a ‘hook’ to run on (OS) operations

  e.g. creating new files

ideal mechanism: OS support

less ideal mechanism: change library loading

  e.g. replace ‘open’, ‘fopen’, etc. in libraries

less ideal mechanism: replace OS exception (system call) handlers

  very OS version dependent
attacking antivirus (2)

just directly modify it
  example: IDEA.6155 modifies database of scanned files

preserve checksums
  example: HybrisF preserved CRC32 checksums of infected files
  some AV software won’t scan again
armored viruses

“encrypted” viruses
not strong encryption — key is there!

self-changing viruses:

encrypted ↔ oligiomorphic ↔ polymorphic ↔ metamorphic

breaking debuggers, antivirus
our model of malware — runs when triggered

reality: sometimes keep on running

evade active detection

spread to new programs/files as created/run
real signatures: ClamAV

ClamAV: open source email scanning software

signature types:

- hash of file
- hash of contents of segment of executable
  - built-in executable, archive file parser
- fixed string
- basic regular expressions
  - wildcards, character classes, alternatives
- more complete regular expressions
  - including features that need more than state machines
- meta-signatures: match if other signatures match
- icon image fuzzy-matching