anti-anti-virus (continued)

logistics: TRICKY

HW assignment out

“infecting” an executable

last time

anti-virus:
- heuristic: malware messing up executables?
  - key idea: wrong segment of executable?
  - switching segments of the executable?
  - weird API calls/names
- detecting bad behavior?

anti-anti-virus
- “encrypted” data
- “encrypted” code: “packers”
- oligomorphic and polymorphic: changing “encrypters”

today

this time:
- metamorphic: changing encrypter and body

also: more anti-anti-virus
- anti-virtualization review
- anti-debugging
- …

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

upcoming assignment: LEX

use `flex` to write a scanner for the pattern:

```
push $0x12345678; ret
```

explain a false positive, propose practical solution

example: 1260 (virus)

```
    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)
    ...
```

```
    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)

```
    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)
    ...
```

```
    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)

```
    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)
    ...
```

```
    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)

```
    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)
    ...
```

```
    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)

```
    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)
    ...
```

```
    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)
    ...
```

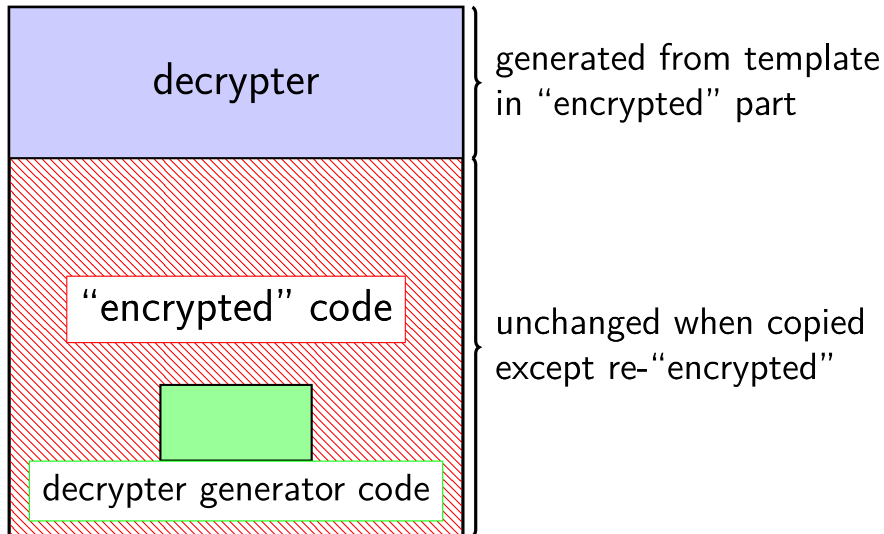
adapted from Szor, Listing 7.5

multiple versions?

in “encrypted” code:

```
void generateDecrypter() {
    int key = random();
    writeRandomNop();
    if (random()) {
        writeMovKey(key);
        writeMovCodeLoc();
    } else {
        writeMovCodeLoc();
        writeMovKey(key);
    }
    writeRandomNop();
    ...
}
```

typical polymorphic malware layout



polymorphic/oligomorphic

only “encrypter” changes

“encrypted” code to generate encrypter

only need to handle what encrypter does

simplest version: just have several versions in encrypted code

second simplest version: template with holes

never need to read machine code!

common theme: run it and see

behavior-based detection:

- detect modifications to system files

defeating encrypters

- run encrypter, look for results

often requires VM

much slower than pattern matching

has its own countermeasures

on goats

analysis — and maybe detection — uses *goat files*

“sacrificial goat” to get changed by malware

easier to look for than patterns in memory?

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

on avoiding goats

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
- ...

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

defeats entire “strategy”

stopping packers

it's unusual to jump to code you wrote

modern OSs: memory is executable *or* writable

very rarely executable and writeable

stopping packers

it's unusual to jump to code you wrote

modern OSs: memory is executable *or* writable

very rarely executable and writeable

diversion: 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)

system calls needed 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

recurring theme

don't analyze code — just run it!

avoids the halting problem, kinda

doesn't matter how much effort spent writing decrypters, etc.

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:

- i.e., emulation or binary translation
- 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?

- FPU instructions
- MMX/SSE instructions
- undocumented (!) CPU instructions

not handling OS features?

- setting up special handlers for segfault
- multithreading
- system calls that make callbacks
- ...

operations not supported

missing instructions?

- 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 virtualization patience

looking for unpacked virus in VM

...or other malicious activity

when are you done looking?

attacking virtualization patience

looking for unpacked virus in VM

...or other malicious activity

when are you done looking?

malware solution: take too long

- not hard if virtualization uses "slow" implementation
- slow? maybe allows more inspection of program (e.g. switching segments, newly written code)

attacking virtualization patience

looking for unpacked virus in VM

…or other malicious activity

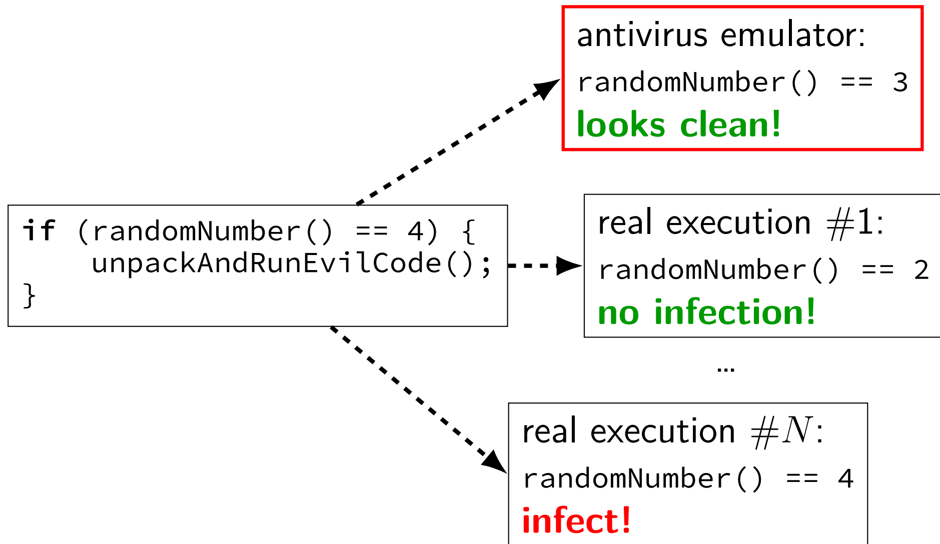
when are you done looking?

malware solution: take too long

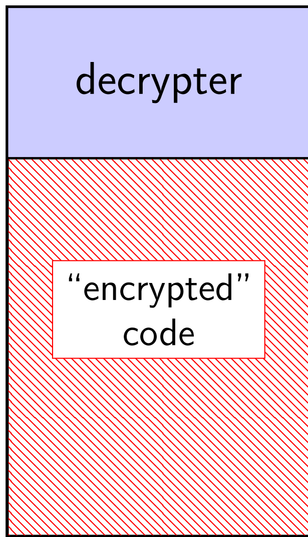
- not hard if virtualization uses “slow” implementation
- slow? maybe allows more inspection of program
(e.g. switching segments, newly written code)

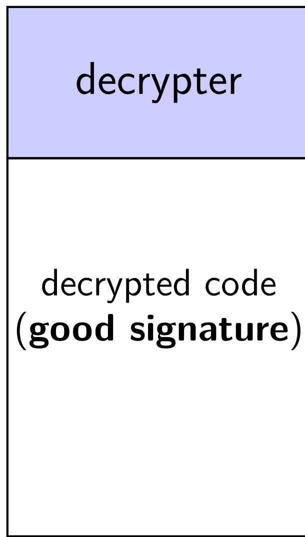
malware solution: don’t infect consistently

probability



signatures in RAM





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"

metamorphic versus polymorphic

big change in difficulty

polymorphic: can have “template” with blanks

metamorphic: probably “understand” machine code

- could have been doing this with polymorphic, but probably not

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,
```

every instruction changes

likely with machine code parser!

- locate register number bits
- swap register numbers w/table lookup

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,
```

still has good signatures

- with alternatives for each possible register selection

but harder to write/slower to match

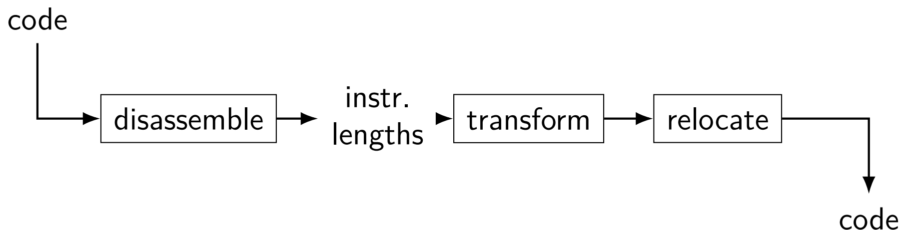
- in addition to running VM to decrypt

case study: Evol

via Lakhatia et al, “Are metamorphic viruses really invincible?”, Virus Bulletin, Jan 2005.

“mutation engine”

run as part of propagating the virus

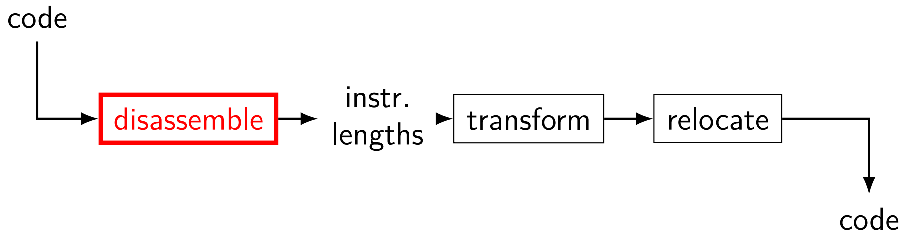


case study: Evol

via Lakhatia et al, "Are metamorphic viruses really invincible?", Virus Bulletin, Jan 2005.

"mutation engine"

run as part of propagating the virus



Evol instruction lengths

sounds really complicated?

- instruction prefixes, ModRM byte parsing, …
- big table of opcodes?

virus only handles instructions it has:

- about 61 opcodes, 32 of them identified by first four bits
- (opcode `0x7`*x* – conditional jump)

no prefixes, no floating point

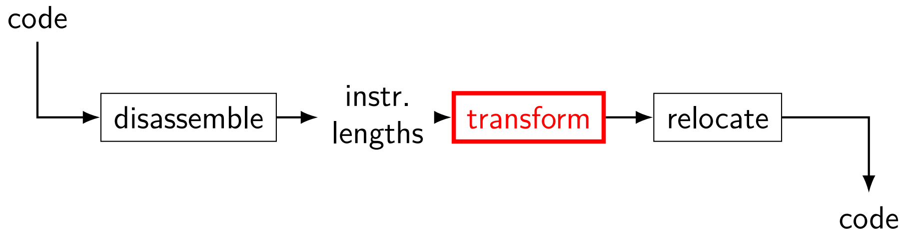
only `%reg` or `$constant` or `offset(%reg)`

case study: Evol

via Lakhatia et al, "Are metamorphic viruses really invincible?", Virus Bulletin, Jan 2005.

"mutation engine"

run as part of propagating the virus



Evol transformations

some stuff left alone

static or random one of N transformations

example:

```
mov %eax, 8(%ebp)
```

```
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

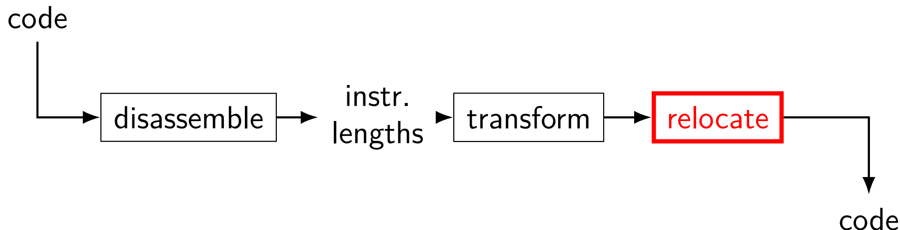
Lakhotia et al., "Are metamorphic viruses really invincible?", Virus Bulletin, Jan 2005

case study: Evol

via Lakhatia et al, “Are metamorphic viruses really invincible?”, Virus Bulletin, Jan 2005.

“mutation engine”

run as part of propagating the virus



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 ...
0x9:
    xor %rax, (%rbx)
    inc %rbx
    dec %rcx
    jne 0x9
```

relocation example

```
    mov ...
    mov ...
0x9:
    xor %rax, (%rbx)
    inc %rbx
    dec %rcx
    jne 0x9
```

table from transformation

orig. loc	new loc	addr	
5	10	--	mov
7	13	--	mov
9	20	--	xor
10	21	--	inc
11	26	--	dec
14	29	9	jne

relocation example

```
    mov ...
    mov ...
0x9:
    xor %rax, (%rbx)
    inc %rbx
    dec %rcx
    jne 0x9
```

table from transformation

orig. loc	new loc	addr	
5	10	--	mov
7	13	--	mov
9	20	--	xor
10	21	--	inc
11	26	--	dec
14	29	9	jne

relocation actions

address loc	orig. target	new target
29+1 (jne1+1)	xor1 (9)	xor1 (20)

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 after any transform)
- apparently done manually
- automatable?

malware authors use 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 machien 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

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 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

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

into

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

problem: `jmp` might be bigger than `addq`?

int 3

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()) {
```

modern debuggers

`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?

```
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 (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?

would take effect on following instruction

…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”

terminology

semistealth/*stealth* — hide from system

tunneling virus — evades behavior-blocking

e.g. detection of modifying system files

retrovirus — directly attacks/disables antivirus software

attacking antivirus (1)

how does antivirus software scan new files?

how does antivirus software detect bad behavior?

- register handlers with OS/applications — new files, etc.

hooking and malware

hooking — getting a 'hook' into (OS) operations

e.g. creating new files, opening file

monitoring or changing/stopping behavior

used by antivirus and malware:

stealth virus — hide virus program from normal I/O, etc.

tunneling virus — skip over antivirus's hook

retrovirus — break antivirus's hook

stealth

```
/* in virus: */
int OpenFile(const char *filename, ...) {
    if (strcmp(filename, "infected.exe") == 0) {
        return RealOpenFile("clean.exe", ...);
    } else {
        return RealOpenFile(filename, ...);
    }
}
```

stealth ideas

override “get file modification time” (infected files)

override “get files in directory” (infected files)

override “read file” (infected files)

but not “execute file”

override “get running processes”

tunneling ideas

use the “real” write/etc. function

not wrapper from antivirus software

find write/etc. function antivirus software “forgot” to hook

retrovirus ideas

empty antivirus signature list

kill antivirus process, remove “hooks”

delete antivirus software, replace with dummy executable

…

hooking mechanisms

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 mechanisms

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

IN THIS ARTICLE +

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 mechanisms

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?

problem both for malware and anti-virus!

hooking mechanisms

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 exception handlers?

mechanism on DOS

track what old exception handler does

“tunneling” technique — find the original, call it instead

other holes in behavior blocking

if in library: don't use library function

- e.g. copy of "clean" library
- e.g. statically linked

generally: multiple ways to do things?

- like VM problem: was something missed?

e.g.. file modifications blocked?

just acccess the disk directly

attacking antivirus (2)

mechanisms other than hooking

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

not just hiding/interfering

our model of malware — runs when triggered

reality: sometimes keep on running

- evade active detection
- spread to new programs/files as created/run

call resident

spreading in memory

hook to hide virus file

not just hiding virus — propogate!

example: infect any new files

example: reinfect “repaired” files

armored viruses

"encrypted" viruses

not strong encryption — key is there!

self-changing viruses:

encrypted → oligiomorphic → polymorphic → metamorphic

anti-debugging, tunnelling, etc.

anti-debugging/virtualisation/goat

evade various "run it and check" techniques

tunnelling

evade behavior-blocking/detection

stealth

"hook" system operations (like antivirus)

hide modified files, malware processes, etc.

retrovirus

deliberately break antivirus software

memory residence

infect running OS/programs, not just files

antivirus needs to kill running virus code

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

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

example heuristics: DREBIN (1)

from 2014 research paper on Android malware: Arp et al, "DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket"

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?