Anti-anti-malware (part 3) / Stack Smashing

last time

- various kinds of armored malware
    - malware that evades analysis

logistics: LEX homework

detect push ret pattern

using flex

probably easier than TRICKY

upcoming exam

next Wednesday

review next Monday — come with questions

armored viruses

"encrypted" malware

- not strong encryption — key is there!

self-changing viruses:

encrypted ◂▸ oligiomorphic ◂▸ polymorphic ◂▸ metamorphic

other anti-analysis techniques:

- antigoat
- antiemulation
- antidebugging

this time

finish up anti-debugging

"tunnelling viruses"

- evade behavior-based detection

memory residence

Nasi article on evading 2014 antivirus

(if time) new topic: exploits and stack-smashing

antiantivirus

last time:

- 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? kinda works

```
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), %r8
```

then `jmp` to `addq`

implementing single-stepping (2)

problem: what about flow control?

```
jmpq *0x1234(%rax,%rbx,8) // ←− STOPPED HERE
```

or

```
retq
```

or

implementing single-stepping (3)

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
- what does the loader really use??

“broken” machine code

- insert “junk” bytes to break disassembly
- skip over junk with jump

use the stack pointer not for the stack

- stack trace?

confusing debuggers

“broken” executable formats

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

recall anti-virus heuristics looking for this
(though brokness probably not on purpose?)

“broken” mac

- insert “junk” bytes to break disassembly
- skip over junk with jump

use the stack pointer not for the stack

- stack trace?

confusing debuggers

“broken” executable formats

- e.g., recall ELF: segments and sections
- corrupt sections — program still works
- overlapping segments/sections — program still works
- what does the loader really use??

“encrypted” code sophisticated version of this

“broken” machine code

- insert “junk” bytes to break disassembly
- skip over junk with jump

use the stack pointer not for the stack

- stack trace?

confusing debuggers

“broken” executable formats

- e.g., recall ELF: segments and sections
- corrupt sections — program still works
- overlapping segments/sections — program still works
- what does the loader really use??

“broken” machine code

- insert “junk” bytes to break disassembly
- skip over junk with jump

use the stack pointer not for the stack

- stack trace?

antiantivirus

last time:

- 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

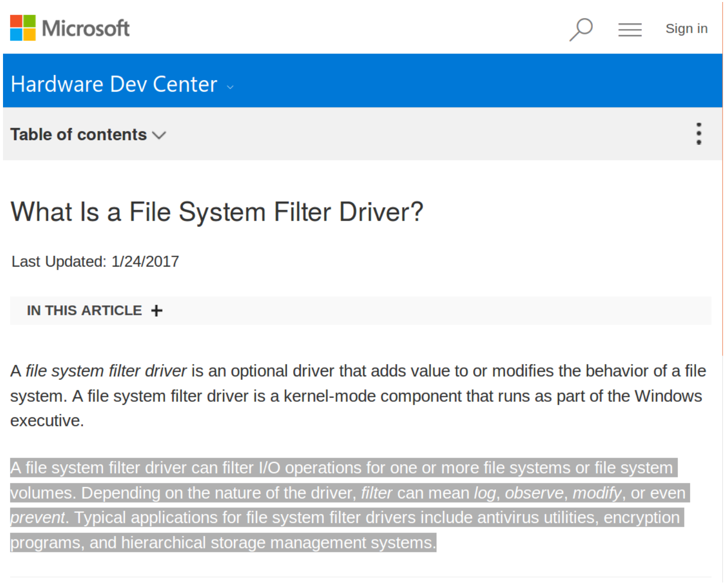
ideal mechanism: OS support

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

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.

debugging mechanisms

debuggers can stop program at system calls, etc.

another form of OS support, typically

Linux interface: `ptrace`

- has "run program until any system call" mode
- and (recently) " run program until specific system call" mode

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?

“anti-virus breaks my program”

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
- solution: use cryptographically secure hashes instead

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 — can propagate!

example: infect any new files

example: reinfect “repaired” files

Emeric Nasi article

Emeric Nasi, "Bypass Antivirus Dynamic Analysis: Limitations of the AV model and how to exploit them", 2014

terminology "FUD = Fully UnDetectable"

NB — not a peer-reviewed article

 "non-traditional literature"

wrote programs, submitted to VirusTotal

 aggregator of antivirus software

looking at detection of new malware

techniques in Nasi that worked

things 2014 Antivirus VM's couldn't handle:

allocate 100 MB

100M increments

un/misimplemented system calls (NUMA, mutex)

check executable name