virus: easiest code to find?

what should be easiest/hardest to identify without many false positives?

A. replaced start location

B. replaced dynamic linker stub

C. replaced dynamic library symbol location

D. replaced function call

E. replaced function return

F. replaced bootloader

G. new automatically started system program

virus choices?

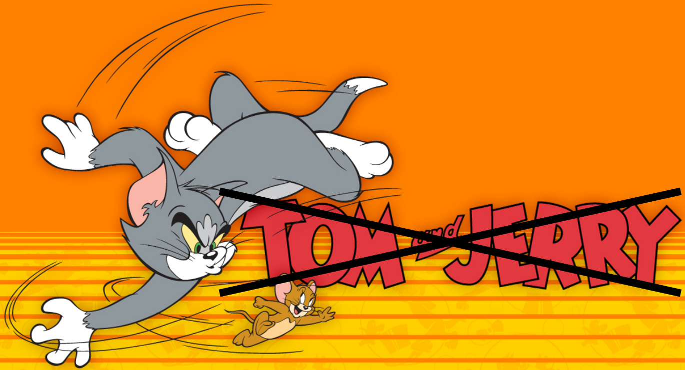
why don't viruses always append/replace?

why don't viruses always change start location?

why did I bother talking about all these strategies?

more on virus strategies

after we talk about anti-virus strategies some



Anti-Virus and Virus

anti-malware software goals

prevent malware from running

prevent malware from spreading

undo the effects of malware

anti-malware software goals

prevent malware from running

prevent malware from spreading

undo the effects of malware

key subproblem: detect malware

tripwire

open source tool from c. 2000

also company around that tool, but I don't know about it

"tool for monitoring and alerting on file & directory changes"

targetted at servers with professional administrators

setup: run tool, it records state of system/etc. files (e.g. hashes)

later: run tool, it tells you if anything changed

tripwire as antimalware software?

tripwire idea: detect any changes

notify user (administrator) about them

what is user supposed to do with this info?

what about normal software updates, etc.?

can malware hide in files that are supposed to change?

- “data” files with other programs, scripts?
- …

what if system compromised before setup?

application whitelisting

how about we only let standard applications run, unmodified?

AppStore-based strategy?

not uncommon in corporate environments:

AppLocker

10/16/2017 • 7 minutes to read • +5

Applies to

- Windows 10
- Windows Server

This topic provides a description of AppLocker and can help you decide if your organization can benefit from deploying AppLocker application control policies. AppLocker helps you control which apps and files users can run. These include executable files, scripts, Windows Installer files, dynamic-link libraries (DLLs), packaged apps, and packaged app installers.

case study: Microsoft AppLocker

AppLocker is Windows 7+ feature for limiting what can run

- successor(?) feature App Control for Business (Windows 10+)

adminstrator sets rules about...

what publisher is allowed

- publisher cryptographically signs applications
- allows easy upgrades! (publisher signs new version)
- virus-like techniques break signatures

what file hashes are allowed

- requires manual update each time software updates

what locations are allowed

- presumably for administrator-only directories

problems with whitelisting

programs with features/bugs malware could exploit

> "AppLocker does not control the behavior of applications after they are launched. Applications could contain flags passed to functions that signal AppLocker to circumvent the rules and allow another .exe or .dll to be loaded."

users might want to install/develop other software

scripting (Python, server-side JavaScript, …):

> "Not all host processes call into AppLocker and, therefore, AppLocker cannot control every kind of interpreted code"

modern bootloaders — secure boot

"Secure Boot" is a common feature of modern bootloaders

idea: UEFI/BIOS code checks bootloader code, fails if not okay

- requires user intervention to use not-okay code

Secure Boot and keys

Secure Boot relies on cryptographic signatures

- idea: accept only "legitimate" bootloaders
- legitimate: known authority vouched for them

user control of their own systems?

- in theory: can add own keys

what about changing OS instead of bootloader?

- bootloader could check cryptographic signature or hash of kernel being loaded

malware “signatures”

typically can’t rely on whitelisting approach

- software and related files change legitimately
- (note: malware might not be in main executables)

antivirus vendor have *signatures* for known malware

many options to represent signatures

thought process: *signature for Vienna?*

goals: compact, fast to check, reliable

aside: signature types

one goal: detect malware without it running

- examine code+data

our first topic

alternate idea: detect running malware

- examine operations performed by software

we'll revisit later

what signature for Vienna?

Suppose we wanted to detect Vienna in execs.

What is best to look for in an exectuable...
in terms of performance? false positives? true positives?

A. machine code found in example infected file at the end of the executable
B. machine code found in example infected file at the end of the executable, ignoring parts that change on reinfection
C. portion of virus's machine code that copies itself to a new file anywhere in the executable
D. whether another executable file in same directory changes if we run the executable in a VM
E. for a jump at beginning of the executable to something near the end

exercise: signatures for Vienna

```
jmp 0x0700 /* C */
mov $0x9e4e, %si /* A */
...              /* A */
/* more app code */
...              /* A */
push %cx
mov $0x8f9, %si /* C */
...
mov $0x0100, %di
mov $3, %cx
rep movsb
...
```

```
...
add $0x2f9, %cx
mov %si, %di
sub $0x1f7, %di
mov %cx, (%di)
...
mov $0x288, %cx
mov $0x40 %ah
mov $si, $dx
sub $0x1f9, %dx
int 0x21
...
```

```
pop %cx
xor %ax, %ax
xor %bx, %bx
xor %dx, %dx
mov $0x0100, %di
push %di
xor %di, %di
ret
/* virus data */
```

/* C */ = constant changes when Vienna relocated

/* A */ = application code

exercise: signatures for Vienna

```
jmp 0x0700 /* C */
mov $0x9e4e, %si /* A */
...              /* A */
/* more app code */
...              /* A */
push %cx
mov $0x8f9, %si /* C */
...
mov $0x0100, %di
mov $3, %cx
rep movsb
...
```

```
...
add $0x2f9, %cx
mov %si, %di
sub $0x1f7, %di
mov %cx, (%di)
...
mov $0x288, %cx
mov $0x40 %ah
mov $si, $dx
sub $0x1f9, %dx
int 0x21
...
```

```
pop %cx
xor %ax, %ax
xor %bx, %bx
xor %dx, %dx
mov $0x0100, %di
push %di
xor %di, %di
ret
/* virus data */
```

/* C */ = constant changes when Vienna relocated

/* A */ = application code

exercise: signatures for Vienna

```
jmp 0x0700 /* C */
mov $0x9e4e, %si /* A */
...              /* A */
/* more app code */
...              /* A */
push %cx
mov $0x8f9, %si /* C */
...
mov $0x0100, %di
mov $3, %cx
rep movsb
...
```

```
...
add $0x2f9, %cx
mov %si, %di
sub $0x1f7, %di
mov %cx, (%di)
...
mov $0x288, %cx
mov $0x40 %ah
mov $si, $dx
sub $0x1f9, %dx
int 0x21
...
```

```
pop %cx
xor %ax, %ax
xor %bx, %bx
xor %dx, %dx
mov $0x0100, %di
push %di
xor %di, %di
ret
/* virus data */
```

/* C */ = constant changes when Vienna relocated

/* A */ = application code

exercise: signatures for Vienna

```
jmp 0x0700 /* C */
mov $0x9e4e, %si /* A */
...              /* A */
/* more app code */
...              /* A */
push %cx
mov $0x8f9, %si /* C */
...
mov $0x0100, %di
mov $3, %cx
rep movsb
...
```

```
...
add $0x2f9, %cx
mov %si, %di
sub $0x1f7, %di
mov %cx, (%di)
...
mov $0x288, %cx
mov $0x40 %ah
mov $si, $dx
sub $0x1f9, %dx
int 0x21
...
```

```
pop %cx
xor %ax, %ax
xor %bx, %bx
xor %dx, %dx
mov $0x0100, %di
push %di
xor %di, %di
ret
/* virus data */
```

/* C */ = constant changes when Vienna relocated

/* A */ = application code

simple signature (1)

all the code Vienna copies

… except changed `mov` to `%si`

virus doesn't change it to relocate

includes infection code — definitely malicious

signature generality

the Vienna virus was copied a bunch of times

small changes, “payloads” added

- print messages, do different malicious things, ...

this signature will not detect any variants

can we do better?

simple signature (2)

Vienna start code

- weird jump at beginning??

problem: maybe real applications do this?

problem: easy to move jump

simple signature (3)

Vienna infection code

- scans directory, finds files

likely to stay the same in variants?

simple signature (3)

Vienna infection code

  scans directory, finds files

likely to stay the same in variants?

problem: virus writers *react to antivirus*

simple signature (4)

Vienna finish code

push + ret

very unusual pattern

probably(?) not in “real” programs

real effort to change to something else?

simple signature (4)

Vienna finish code

push + ret

very unusual pattern

probably(?) not in "real" programs

real effort to change to something else?

problem: virus writers *react to antivirus*

making things hard for the mouse

don't want *trivial changes* to break detection

want to detect *strategies*

e.g. require changing relocation logic

…not just reordering instructions, adding nops

need to detect signatures in real time

don't want interrupt user (much)

want to avoid false positive

goals: compact, fast to check, reliable, *general?*

generic pattern example

another possibility: detect writing near `0x100`

`0x100` was DOS program entry code — no program should do this(?)

problem: how to represent this?

- describe machine code bytes
- multiple possibilities

regular expressions

one method of representing patterns like this:
regular expressions (regexes)

restricted language allows very fast implementations

especially when there's a long list of patterns to look for

upcoming homework assignment

regular expressions: implementations

multiple implementations of regular expressions

we will target: flex, a parser generator

simple patterns

alphanumeric characters *match themselves*

`foo`:

- matches exactly `foo` only
- does not match `Foo`
- does not match `foo␣`
- does not match `foobar`

backslash might be needed for others

`C\+\+`

- matches exactly `C++` only

metachars (1)

special ways to match characters

`\n`, `\t`, `\x3C`, …— work like in C

`[b-fi]` — `b` or `c` or `d` or `e` or `f` or `i`

`[^b-fi]` — any character but `b` or `c` or …

`.` — any character except newline

`(.|\n)` — any character

metachars (2)

`a*` — zero or more as:

(empty string), a, aa, aaa, ...

`a{3,5}` — three to five as:

aaa, aaaa, aaaaa

`(abc){3,5}` — three to five abcs: (“grouping”)

abcabcabc, abcabcabcabc, abcabcabcabcabc

`ab|cd`

ab, cd

`(ab|cd){2}` — two ab-or-cds:

abab, abcd, cdab, cdcd

metachars (3)

`\xAB` — the byte `0xAB`

`\x00` — the byte `0x00`

flex is designed for text, handles binary fine

`\n` — newline (and other C string escapes)

example regular expressions

match words ending with `ing`:

```
[a-zA-Z]*ing
```

match C `/* ... */` comments:

```
/\*([^*]|\*[^/])*\*/
```

flex

flex is a regular expression matching tool

intended for writing *parsers*

generates *C code*

parser function called `yylex`

flex example

```
        int num_bytes = 0, num_lines = 0;
        int num_foos = 0;
%%
foo     {
          num_bytes += 3;
          num_foos += 1;
        }
.       { num_bytes += 1; }
\n      { num_lines += 1; num_bytes += 1; }
%%
int main(void) {
    yylex();
    printf("%d bytes, %d lines, %d foos\n",
           num_bytes, num_lines, num_foos);
```

flex example

```
        int num_bytes = 0, num_lines = 0;
        int num_foos = 0;
%%
foo     {
          num_bytes += 3;
          num_foos += 1;
        }
.       { num_bytes += 1; }
\n      { num_lines += 1; num_bytes += 1; }
%%
int main(void) {
    yylex();
    printf("%d bytes, %d lines, %d foos\n",
           num_bytes, num_lines, num_foos);
```

three sections

flex example

```
        int num_bytes = 0, num_lines = 0;
        int num_foos = 0;
%%
foo     {
          num_bytes += 3;
          num_foos
        }
.       { num_bytes += 1; }
\n      { num_lines += 1; num_bytes += 1; }
%%
int main(void) {
    yylex();
    printf("%d bytes, %d lines, %d foos\n",
           num_bytes, num_lines, num_foos);
```

first — declarations for later C code in output file

flex example

patterns, code to run on match
as parser: return “token” here

```
        int num_bytes = 0, num_lines = 0;
        int num_fo
%%
foo     {
          num_bytes += 3;
          num_foos += 1;
        }
.       { num_bytes += 1; }
\n      { num_lines += 1; num_bytes += 1; }
%%
int main(void) {
    yylex();
    printf("%d bytes, %d lines, %d foos\n",
           num_bytes, num_lines, num_foos);
```

flex example

```
        int num_bytes = 0, num_lines = 0;
        int num_foos = 0;
%%
foo     {
          num_bytes += 3;
          num_foos +=
        }
.       { num_bytes += 1; }
\n      { num_lines += 1; num_bytes += 1; }
%%
int main(void) {
    yylex();
    printf("%d bytes, %d lines, %d foos\n",
           num_bytes, num_lines, num_foos);
```

extra code to include

flex: matched text

```
%%
[aA][a-z]* {
                printf("found a-word '%s'\n",
                       yytext);
           }
(.|\n)     {} /* default rule: would output text */
%%
int main(void) {
    yylex();
}
```

flex: matched text

yytext — text of matched thing

```
%%
[aA][a-z]* {
                printf("found a-word '%s'\n",
                       yytext);
           }
(.|\n)     {} /* default rule: would output text */
%%
int main(void) {
    yylex();
}
```

flex: definitions

```
A           [aA]
LOWERS      [a-z]
ANY         (.|\n)
%%
{A}{LOWERS}* {
                printf("found a-word '%s'\n",
                       yytext);
             }
{ANY}        {} /* default rule would
                   output text */
%%
int main(void) {
    yylex();
}
```

flex: definitions

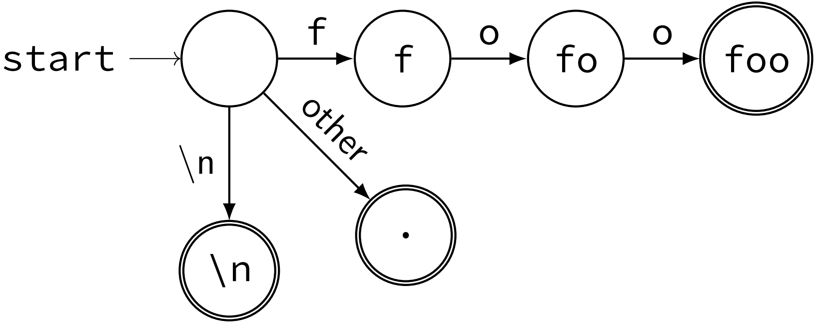
```
A           [aA]
LOWERS      [a-z]
ANY         (.|\n)
%%
{A}{LOWERS}* {
                 printf("found a-word '%s'\n",
                        yytext);
             }
{ANY}        {} /* default rule would
                   output text */

%%
int main(void) {
    yylex();
}
```

definitions of common patterns included later

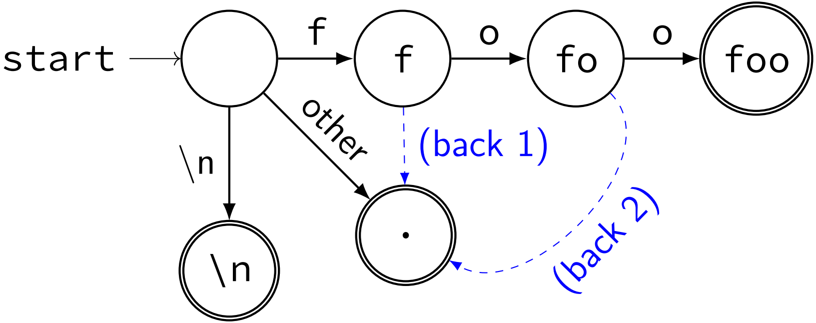
flex: state machines

```
foo      {...}
.        {...}
\n       {...}
```



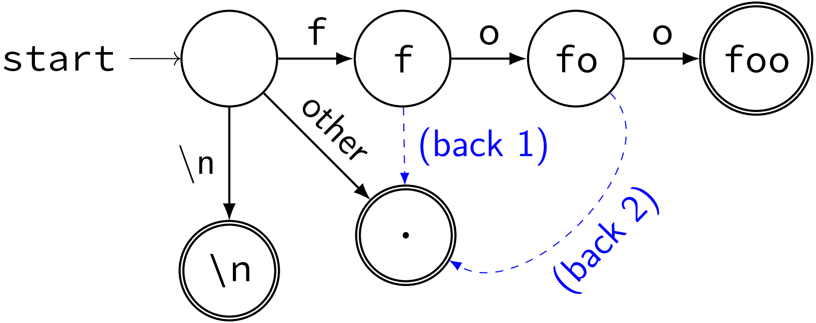
flex: state machines

```
foo      {...}
.        {...}
\n       {...}
```



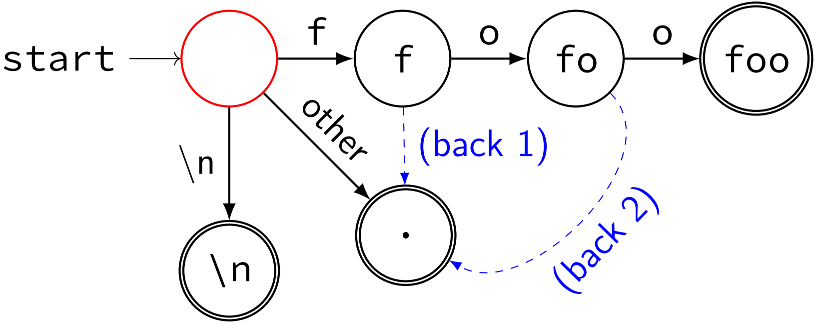
state machine matching

abfoofoabffoo



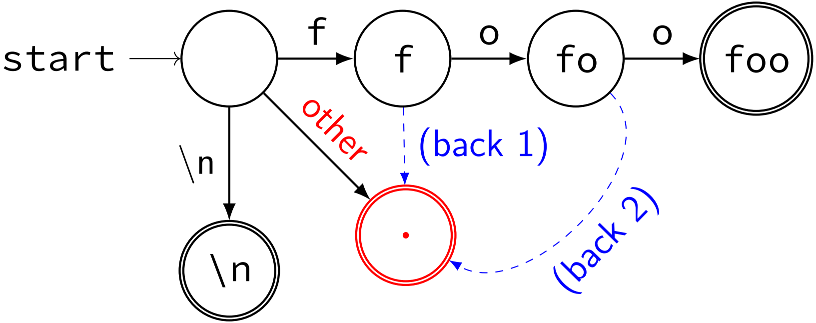
state machine matching

*a*bfoofoabffoo



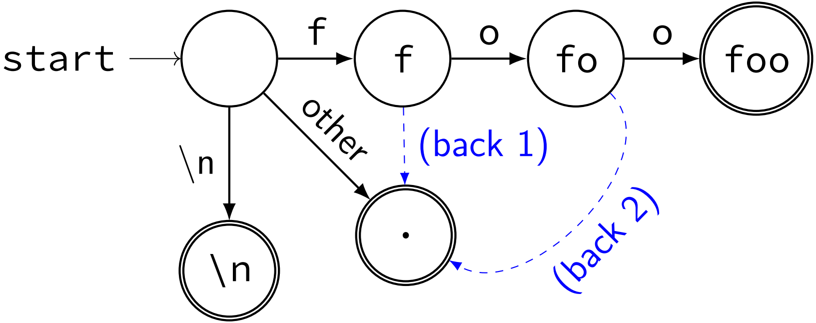
state machine matching

~~a~~bfoofoabffoo



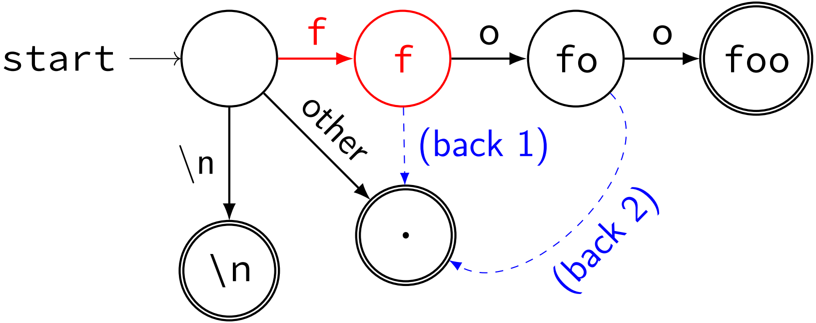
state machine matching

~~ab~~foofoabffoo



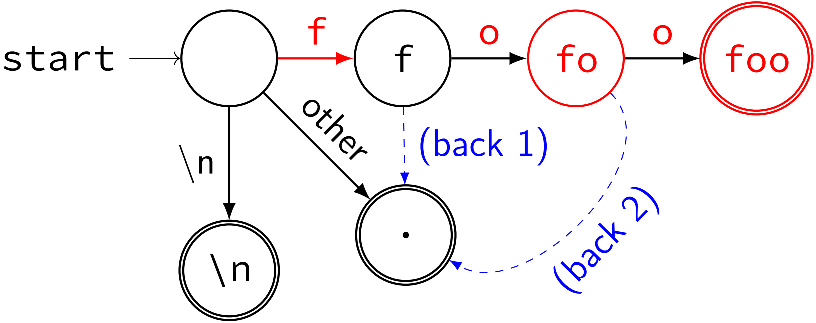
state machine matching

~~ab~~foofoabffoo



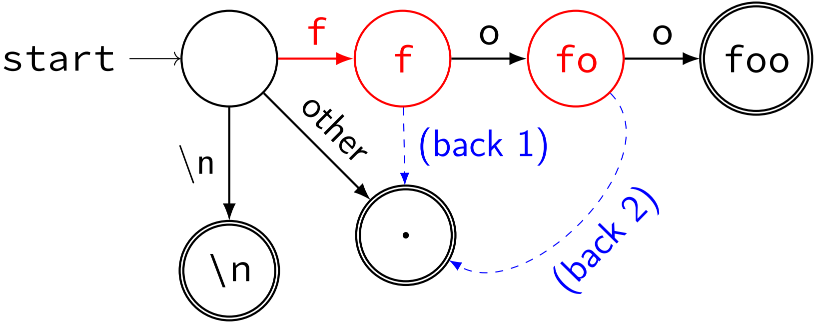
state machine matching

~~ab~~*foo*foabffoo



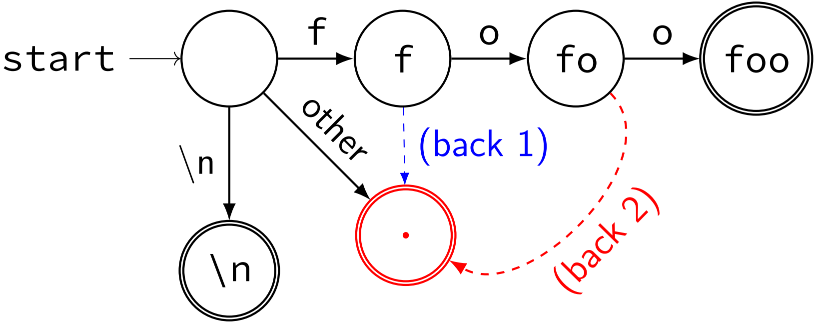
state machine matching

~~abfoo~~*fo*abffoo



state machine matching

~~abfoo~~*f*~~o~~abffoo



why this?

one pass matching (except for some backtracking)

can make state machine bigger to avoid some backtracking

basically speed of file I/O

handles multiple patterns well

flexible for “special cases”

why this?

one pass matching (except for some backtracking)

- can make state machine bigger to avoid some backtracking

basically speed of file I/O

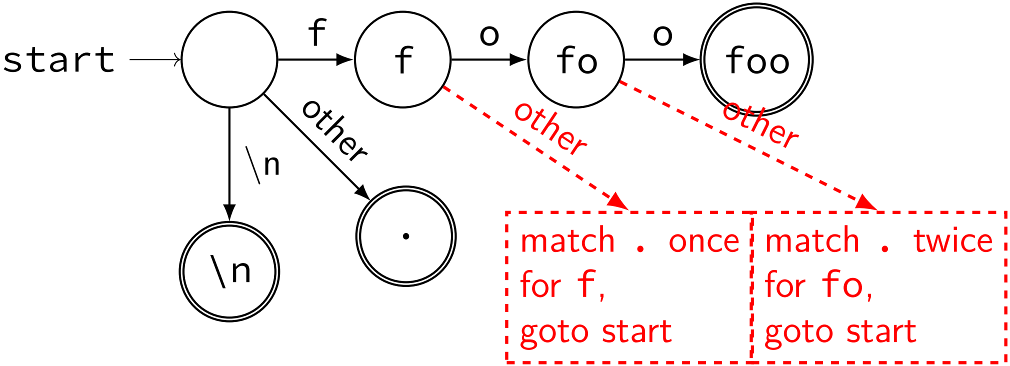
handles multiple patterns well

flexible for "special cases"

real anti-virus: probably custom pattern "engine"

precomputing backtracking

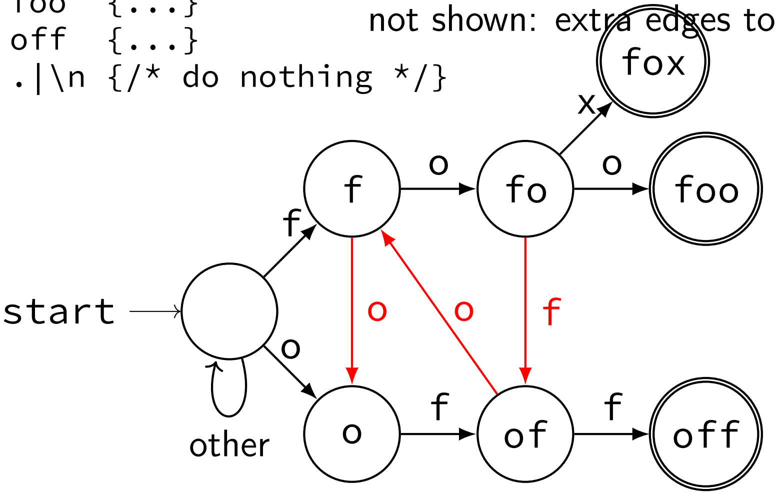
```
foo      {...}
.        {...}
\n       {...}
```



avoiding backtracking?

```
fox  {...}
foo  {...}
off  {...}
.|\n {/* do nothing */}
```

not shown: extra edges to start



Vienna patterns (1)

simple Vienna patterns:

```
/* bytes of fixed part of Vienna sample */
\xFC\x89\xD6\x83\xC6\x81\xc7\x00\x01\x83(etc) {
        printf("found Vienna code\n");
    }
```

Vienna patterns (2)

simple Vienna patterns:

```
/* Vienna sample with wildcards for
   changing bytes: */
/* push %CX; mov ???, %dx; cld; ... */
\x51\xBA(.|\n)(.|\n)\xFC\x89(etc) {
        printf("found Vienna code w/placeholder\n");
    }
/* mov $0x100, %di; push %di; xor %di, %di; ret */
\xBF\x00\x01\x57\x31\xFF\xC3 {
        printf("found Vienna return code\n");
    }
```

Vienna patterns (2)

simple Vienna patterns:

```
/* Vienna sample with wildcards for
   changing bytes: */
/* push %CX; mov ???, %dx; cld; ... */
\x51\xBA(.|\n)(.|\n)\xFC\x89(etc) {
        printf("found Vienna code w/placeholder\n");
    }
/* mov $0x100, %di; push %di; xor %di, %di; ret */
\xBF\x00\x01\x57\x31\xFF\xC3 {
        printf("found Vienna return code\n");
    }
```

regular expressions are flexible

for Vienna: lots of flex features we didn't need

- things being repeated variable number of times
- one of list of possible characters (bytes)
- …

but viruses try to make pattern matching hard

good to think about what we can easily match

hard for patterns?

malware makes modificates to evade pattern matching

exercise: suppose we have a pattern for a Vienna-like virus, and a new version makes the following change. Which of the following is going to be easiest/hardest to change the pattern for?

A. inserting random number of nops every 8 non-nop instructions of virus code
B. replacing code at random offset in executable instead of appending
C. registers used for temporaries in virus code chosen at random each time the virus copies itself
D. instead of appending all the virus code, virus code now split between cavities with a "loader" appended (the "loader" reforms code from the cavities and jumps to them)

making scanners efficient

lots of viruses!

- huge number of states, tables
- copies of every piece of malware pretty large

reading files is slow!

making scanners efficient

lots of viruses!

- huge number of states, tables
- copies of every piece of malware pretty large

reading files is slow!

handling volume

storing signature strings is non-trivial

tens of thousands of states???

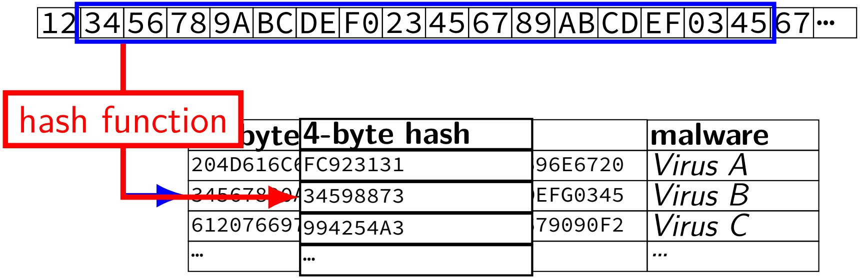
observation: fixed strings dominate

scanning for fixed strings

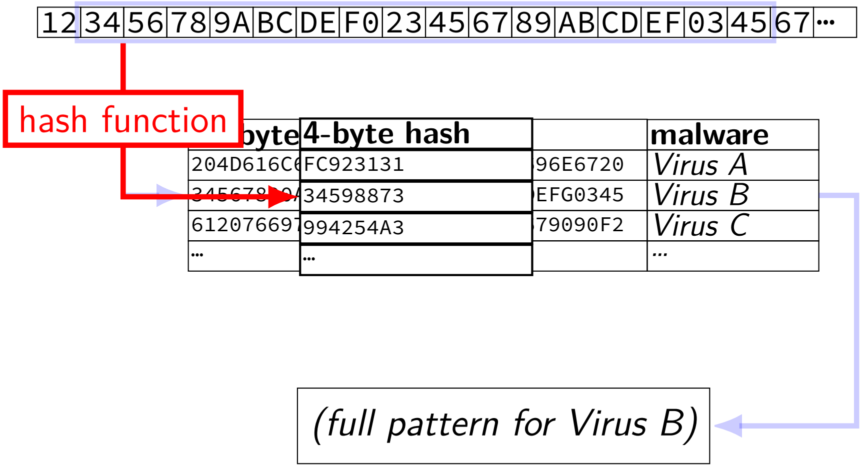
12	34	56	78	9A	BC	DE	F0	23	45	67	89	AB	CD	EF	03	45	67	…

16-byte "anchor"	**malware**
`204D616C6963696F7573205468696E6720`	*Virus A*
`34567890ABCDEF023456789ABCDEFG0345`	*Virus B*
`612076697275732073747269696E679090F2`	*Virus C*
…	…

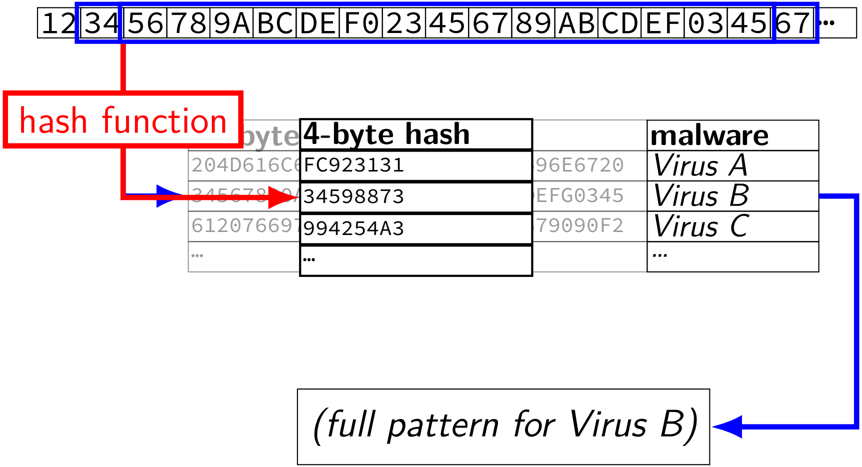
scanning for fixed strings



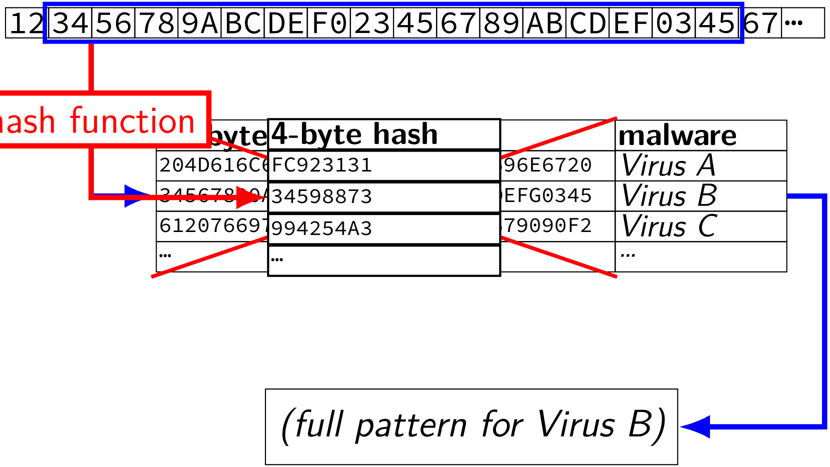
scanning for fixed strings



scanning for fixed strings



scanning for fixed strings



making scanners efficient

lots of viruses!

- huge number of states, tables
- copies of every piece of malware pretty large

reading files is slow!

the I/O problem

scanning still requires reading the whole file

can we do better?

selective scanning

check entry point and end only

- a lot less I/O, maybe

check known offsets from entry point

heuristic: is entry point close to end of file?

real signatures: ClamAV

ClamAV: open source (mostly 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

example ClamAV signatures (1)

hashes

```
4b3858c8b35e964a5eb0e291ff69ced6:78454:Xls.Exploit.Agent-4323916-1:73
7873be8fc5e052caa70fdb8f76205892:293376:Win.Trojan.Sality-93158:73
f358d77926045cba19131717a7b15dec:293376:Win.Trojan.Sality-93159:73
48d4c5294357e664bac1a07fce82ea22:450024:Win.Trojan.Sality-93160:73
e4b8442638b3948ab0291447affa6790:293376:Win.Trojan.Sality-93161:73
df36dc207b689a73ab9cf45a06fb71b0:232448:Win.Trojan.Sality-93162:73
baaeeabc7f4be3199af3d82d10c6b39f:293376:Win.Trojan.Sality-93163:73
...
```

example ClamAV signatures (2)

simple regular expressions (with hex, different syntax than flex)...

```
Win.Trojan.Vienna-1:0:*:5051e8??00{1-255}5b83eb??fc8d37bf0001b90300f3a48bf3558bec8
Win.Trojan.Vienna-2:0:*:be000356c3*50be????8bd6fcb90500bf0001f3a48bfab430cd21
Win.Trojan.Vienna-3:0:*:50ba????8bf283c60090bf0001b90300fcf3a48bfab430cd213c02
Win.Trojan.Vienna-4:0:*:b440b900048bd681eac102cd21721f3d
Win.Trojan.Vienna-5:0:*:b904048bd681ea130352515350b4
...
Win.Trojan.Vienna-129:0:*:51b89b03cd213d01017503e9????ba6d03fc8bf283c60a90b90300bf
```

example ClamAV signatures (3)

'logical' signatures: mutliple regexes together:

```
Andr.Trojan.Pjapps-58;Engine:51-255,
    Container:CL_TYPE_ZIP,Target:0;
    (6&0&1&(2|3)&(4|5));  // expected patterns of below
    3a39303333; // pattern 0
    696d6569;   // pattern 1
    616e64726f69642e6c6f67; // pattern 2
    77696e646f772e6c6f67;   // pattern 3
    4e6f6b69614e373631302d31; // pattern 4
    336c676f6167646d66656a656b67666f733974313563686f6a6d; // pattern 5
    0:646578 // pattern 6: "0:" means must be found at beginning of file
```

playing cat

harder to fool ways of detecting malware?

goal: small changes to malware preserve detection

ideal:

- detect *new* malware
- detect things malware needs to do accomplish their goals

detecting new malware

look for anomalies

- patterns of code that real executables "won't" have

identify bad behavior

viruses and executable formats

header: machine type, file type, etc.
program header: "*segments*" to load (also, some other information)
segment 1 data
segment 2 data

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?***

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?***

heuristic 1: is entry point in last segment?
(segment usually not code)

viruses and executable formats

header: machine type, file type, etc.
program header: "*segments*" to load (also, some other information) ***new segment added by virus***
segment 1 data
segment 2 data
segment 3 data — virus segment

viruses and executable formats

header: machine type, file type, etc.
program header: "*segments*" to load (also, some other information) ***new segment added by virus***
segment 1 data
segment 2 data
segment 3 data — virus segment

heuristic 1: is entry point in last segment?
(segment usually not code)

viruses and executable formats

header: machine type, file type, etc.
program header: "*segments*" to load (also, some other information) ***new segment added by virus***
segment 1 data
segment 2 data
segment 3 data — virus segment

heuristic 2: did virus mess up header?
(e.g. do sizes used by linker but not loader disagree)
section names disagree with usage?

defeating entry point checking

insert jump in normal code section, set as entry-point

add code to first section instead (perhaps insert new section at beginning)

defeating entry point checking

insert jump in normal code section, set as entry-point

add code to first section instead (perhaps insert new section at beginning)

“dynamic” heuristic: run code in VM, 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

evading library call checking

modify dynamic linking tables

- probably tricky to add new entry

reimplement library call manually

- Windows system calls not well documented, change

hide names

evading library call checking

modify dynamic linking tables

- probably tricky to add new entry

reimplement library call manually

- 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

behavior-based detection

things malware does that other programs don't?

basic idea: run in virtual machine; and/or monitor all programs

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

...

basic idea: run in virtual machine; and/or monitor all programs

hooking

hooking — getting a 'hook' to run on (OS) operations

- e.g. creating new files
- e.g. modifying executable 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

less ideal mechanism: debugger support

hooking

hooking — getting a 'hook' to run on (OS) operations

- e.g. creating new files
- e.g. modifying executable 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

less ideal mechanism: debugger support

Microsoft Sign in

Hardware Dev Center

Table of contents

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.

Linux hooking

several possible mechanisms

tracepoints, kprobes

- cause hooking functions to run when kernel functions called or return
- hooker function can arrange for logging or other action

seccomp BPF

- allow hooker to write 'program' to examine system calls of selected processes
- can deny/change/log those system calls

aside Linux eBPF

eBPF = extended Berkeley Packet Filters

little programming language originally intended for network filtering

hooking

hooking — getting a ‘hook’ to run on (OS) operations

- e.g. creating new files
- e.g. modifying executable 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

less ideal mechanism: debugger support

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

hooking — getting a 'hook' to run on (OS) operations

- e.g. creating new files
- e.g. modifying executable 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

less ideal mechanism: debugger support

changing exception call handlers (1)

OS data structure tells hardware where program requests go

simpliest mechanism: edit that data structure

- and save a copy of what was there before

point to your code

- and call what was there before after behavior check

heuristics example: DREBIN paper

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

primary contribution of paper: big dataset of malware

but tried to detect malware, too...

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

heuristics example: DREBIN paper

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?

machine learning and adversaries

I don't like most ML-based approaches to malware detection

problem: most machine learning not designed to deal with adversaries

attack: find factors used to ID benign programs
- do all of them as much as possible
- inquiry: what might they be in DREBIN case?

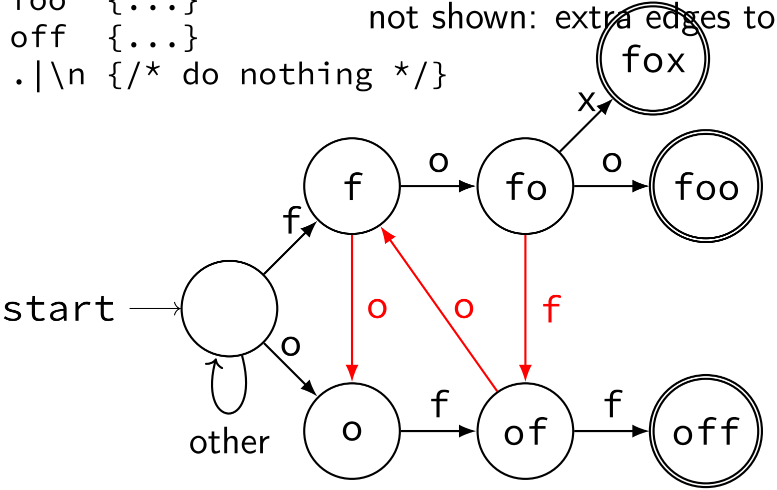
attack: provide many malware samples with benign weird behavior
- machine learning uses weird behavior to identify malware
- may lower effectiveness on 'normal' malware

backup slides

avoiding backtracking?

```
fox   {...}
foo   {...}
off   {...}
.|\n {/* do nothing */}
```

not shown: extra edges to start



flex states (1)

```
%x str
%%
\"              { BEGIN(str); }
<str>\"         { BEGIN(INITIAL); }
<str>foo        { printf("foo in string\n"); }
foo             { printf("foo out of string\n"); }
<INITIAL,str>(.|\n) {}
%%
int main(void) {
    yylex();
}
```

flex states (1)

```
%x str
%%
\"            { BEGIN(str); }
<str>\"       { BEGIN(INITIAL); }
<str>foo      { printf("foo in string\n"); }
foo           { printf("foo out of string\n"); }
<INITIAL,str>(.|\n) {}
%%
int ma
    yy
}
```

declare “state” to track

which state determines what patterns are active

flex states (1)

```
%x str
%%
\"              { BEGIN(str); }
<str>\"         { BEGIN(INITIAL); }
<str>foo        { printf("foo in string\n"); }
foo             { printf("foo out of string\n"); }
<INITIAL,str>(.|\n) {}
%%
int main(void) {
    yylex();
}
```

flex states (2)

```
%s afterFoo
%%
<afterFoo>foo    { printf("later␣foo\n"); }
foo              {
                   printf("first␣foo\n");
                   BEGIN(afterfoo);
                 }
(.|\n)  {}
%%
int main(void) {
    yylex();
}
```

flex states (2)

```
%s afterFoo
%%
<afterFoo>foo     { printf("later␣foo\n"); }
foo               {
                    printf("first␣foo\n");
                    BEGIN(afterfoo);
                  }
(.|\n)  {}
%%
int main(void) {
    yylex();
}
```

declare non-exclusive state

handling packers

easiest way to decrypt self-decrypting code — run it!

solution: *virtual machine/emulator/debugger* in antivirus software

handling packers with debugger/emulator/VM

run program in debugger/emulator/VM for a while

- one heuristic: until it jumps to written data

example implementation: unipacker (`https://github.com/unipacker/unipacker`)

then dump memory to get decrypted machine code

and/or obtain trace of instructions run

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

rootkits

rootkit — priviliged malware that hides itself

same ideas as these anti-anti-virus techniques

rootkits and whitelisting

talked about application whitelisting

- only “known” code authors
- only certain list of applications

was problematic when users want to run lots of applications

rootkits and whitelisting

talked about application whitelisting

- only "known" code authors
- only certain list of applications

was problematic when users want to run lots of applications

users less likely to run software that needs access to 'hook' OS

Windows driver signing

ⓘ Note

Starting with Windows 10, version 1607, Windows will not load any new kernel-mode drivers which are not signed by the Dev Portal. To get your driver signed, first Register for the Windows Hardware Dev Center program. Note that an EV code signing certificate is required to establish a dashboard account.

There are many different ways to submit drivers to the portal. For production drivers, you should submit HLK/HCK test logs, as described below. For testing on Windows 10 client only systems, you can submit your drivers for attestation signing, which does not require HLK testing. Or, you can submit your driver for Test signing as described on the Create a new hardware submission page.

Window driver key stealing

Microsoft signing keys keep getting hijacked, to the delight of Chinese threat actors

What's the point of locks when hackers can easily get the keys to unlock them?

DAN GOODIN – AUG 25, 2023 9:17 AM 76

aside: driver or not driver?

why does random device driver have permission to do all these 'hiding' operations?

(if you've taken CSO2) kernel mode $\rightarrow$ full hardware access

there are OS designs where drivers don't run with full access

- but real performance/complexity costs

chkrootkit

`chkrootkit` — Unix program that looks for rootkit signs

tell-tale strings in system programs

    e.g. file, process, network connection listing programs changed

disagreement between process list, other ways of detecting processes

disagreement between file lists, other ways of counting files

overwritten entries in system login records

known suspicious filenames

    hidden exes in temporary, data directories, etc.

after scanning — disinfection

antivirus software wants to *repair*

requires specialized scanning

- no room for errors
- need to identify *all*
- need to find relocated bits of code