viruses 3 / anti-virus

Changelog

Corrections made in this version not in first posting:

8 Feb 2017: slide 31: visible space after negative foo example

8 Feb 2017: slide 35: `[a-zA-Z]*ing` instead of `[a-zA-Z]ing`

8 Feb 2017: slide 56: correct animation to show hashes second

on due dates

ASM assignment questions?

last time

places to put malicious code

- replace executable
- append/prepend
- cavities
- bootloaders/OS code

started: ways to get code to run

- replace start address
- replace instructions that are run
  - identify returns/function calls/etc.

last time

places to put malicious code
- replace executable
- append/prepend
- cavities
- bootloaders/OS code

started: ways to get code to run
- replace start address
- replace instructions that are run
  - identify returns/function calls/etc.

invoking virus code: options

boot loader

change starting location

alternative approaches: “entry point obscuring”

edit code that’s going to run anyways

replace a function pointer (or similar)

…

run anyways?

add code at start of program (Vienna)

return with padding after it:

```
404a01:      c3                          retq
404a02:      0f 1f 40 00                 nopl   0x0(%rax)
             replace with
404a01:      e9 XX XX XX XX              jmpq    YYYYYYY
```

any random place in program?

- just not in the middle of instruction

recall: finding function calls

e.g. some popular compilers started x86-32 functions with

```
foo:
    push %ebp          // push old frame pointer
    // 0x55
    mov %esp, %ebp     // set frame pointer to stack
    // 0x89 0xec
```

use to identify when `e8` (`call` opcode) refers to real function

(full version: also have some other function start patterns)

remember stubs?

```
0000000000400400 <puts@plt>:
  400400:        ff 25 12 0c 20 00            jmpq   *0x200c12(%rip)
                    /* 0x200c12+RIP = _GLOBAL_OFFSET_TABLE_+0x18 */
  400406:        68 00 00 00 00               pushq  $0x0
  40040b:        e9 e0 ff ff ff               jmpq   4003f0 <_init+0x28>
     replace with:
  400400:        e8 XX XX XX XX           jmpq virus_code
  400405:       90                       nop
  400406:        68 00 00 00 00               pushq  $0x0
  40040b:        e9 e0 ff ff ff               jmpq   4003f0 <_init+0x28>
```

in known location (particular section of executable)

dynamic linker: just modifies global offset table

invoking virus code: options

boot loader

change starting location

alternative approaches: “entry point obscuring”

edit code that’s going to run anyways

replace a function pointer (or similar)

…

stubs again

```
0000000000400400 <puts@plt>:
  400400:       ff 25 12 0c 20 00         jmpq   *0x200c12(%rip)
                   /* 0x200c12+RIP = _GLOBAL_OFFSET_TABLE_+0x18 */
  400406:       68 00 00 00 00            pushq  $0x0
  40040b:       e9 e0 ff ff ff            jmpq   4003f0 <_init+0x28>
```

don't edit stub — edit initial value of `_GLOBAL_OFFSET_TABLE`

stored in data section of executable

originally: pointer to `0x400406`; new — pointer to virus code

virus can `jmp` back to `0x400406` when done

relocations?

```
hello.exe:     file format elf64-x86-64

DYNAMIC RELOCATION RECORDS
OFFSET           TYPE              VALUE
0000000000600ff8 R_X86_64_GLOB_DAT  __gmon_start__
0000000000601018 R_X86_64_JUMP_SLOT  puts@GLIBC_2.2.5
    replace with:
0000000000601018 R_X86_64_JUMP_SLOT  _start + offset_of_virus
0000000000601020 R_X86_64_JUMP_SLOT  __libc_start_main@GLIBC_2.2.5
```

tricky — usually no symbols from executable in dynamic symbol table

- (debugger/disassembler symbols are different tables)
- Linux — need to link with `-rdynamic`

relocations?

```
hello.exe:     file format elf64-x86-64

DYNAMIC RELOCATION RECORDS
OFFSET           TYPE              VALUE
0000000000600ff8 R_X86_64_GLOB_DAT  __gmon_start__
0000000000601018 R_X86_64_JUMP_SLOT  puts@GLIBC_2.2.5
    replace with:
0000000000601018 R_X86_64_JUMP_SLOT  _start + offset_of_virus
0000000000601020 R_X86_64_JUMP_SLOT  __libc_start_main@GLIBC_2.2.5
```

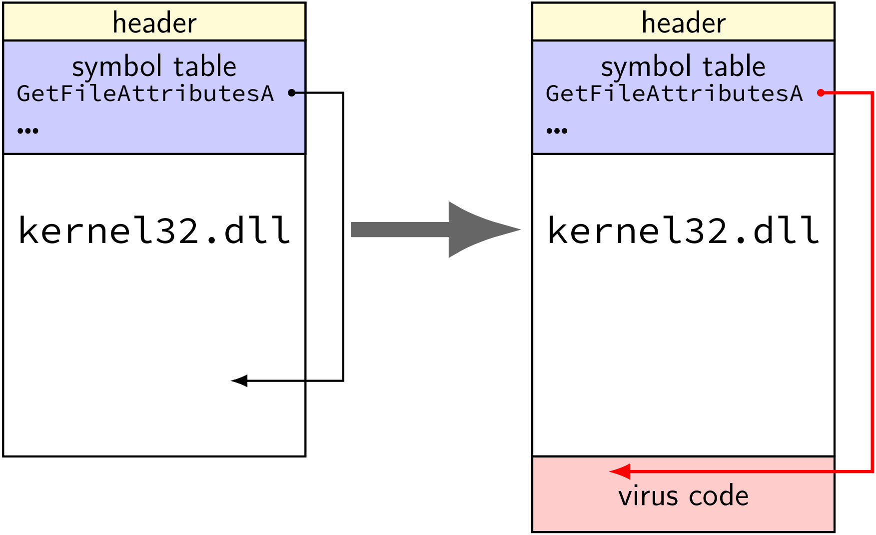
tricky — usually no symbols from executable in dynamic symbol table

(debugger/disassembler symbols are different tables)

Linux — need to link with `-rdynamic`

but…same idea works on shared library itself

infecting shared libraries



TRICKY

next assignment: TRICKY

insert "tricky jump" to virus code

- replacing "`ret`" followed by cavity of `nop`s

submission: program to modify supplied executable

- need not work on any other program
- but, question: how you'd modify it to work on other programs

virus choices?

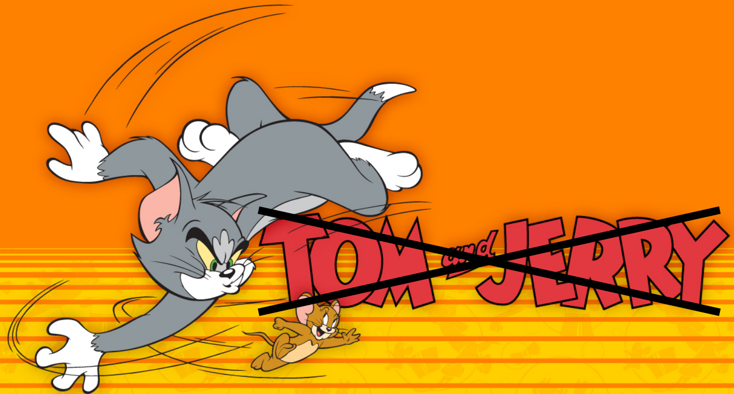
why don't viruses always append/replace?

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 strategies

antivirus goals:

- prevent malware from running
- prevent malware from spreading
- undo the effects of malware

malware detection

important part: detecting malware

simple way:

- have a copy of a malicious executable
- compare every program to it

malware detection

important part: detecting malware

simple way:

- have a copy of a malicious executable
- compare every program to it

how big? every executable infected with every virus?

malware detection

important part: detecting malware

simple way:

- have a copy of a malicious executable
- compare every program to it

when? how fast?

malware "signatures"

antivirus vendor have signatures for known malware

many options to represent signatures

thought process: signature for Vienna?

goals: compact, fast to check, reliable

exercise: signatures for Vienna

```
jmp 0x0700
mov $0x9e4e, %si
...
/* app code */
...
push %cx
mov $0x8f9, %si
...
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 */
```

exercise: signatures for Vienna

```
jmp 0x0700
mov $0x9e4e, %si
...
/* app code */
...
push %cx
mov $0x8f9, %si
...
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 */
```

exercise: signatures for Vienna

```
jmp 0x0700
mov $0x9e4e, %si
...
/* app code */
...
push %cx
mov $0x8f9, %si
...
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 */
```

exercise: signatures for Vienna

```
jmp 0x0700
mov $0x9e4e, %si
...
/* app code */
...
push %cx
mov $0x8f9, %si
...
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 */
```

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

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

signature checking

how fast is signature checking?

problem: lots of I/O?

problem: how complicated are signatures?

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

homework assignment next week

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

```
        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);
}
```

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

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);
}
```

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();
}
```

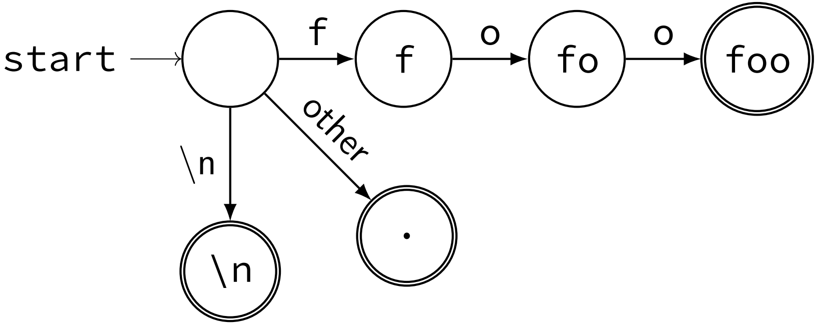
flex: definitions

```
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

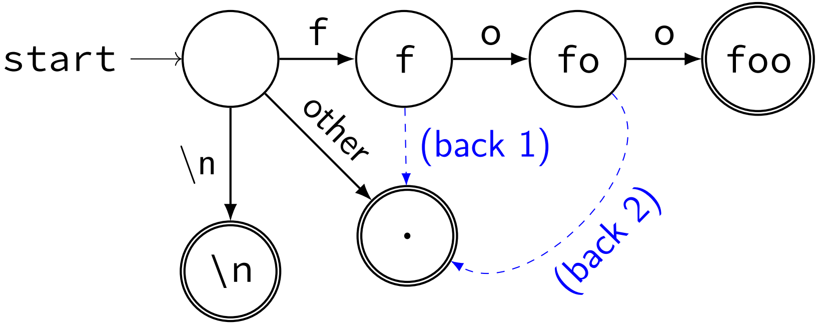
flex: state machines

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



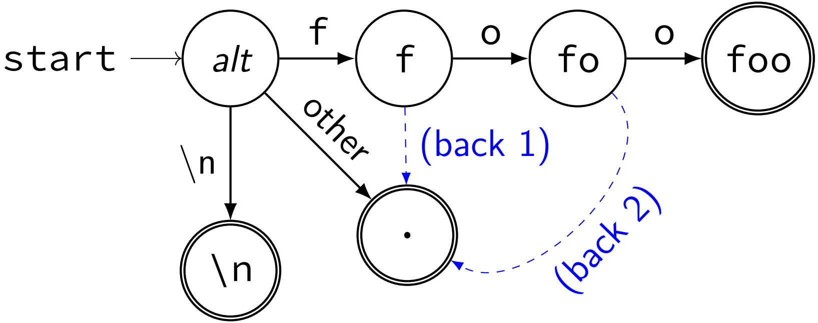
flex: state machines

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



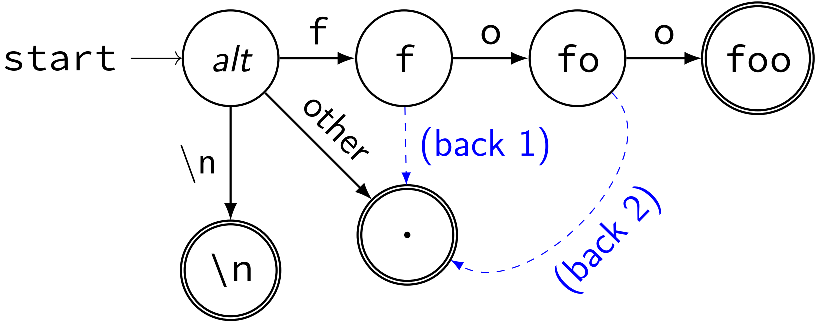
state machine matching

abfoofoabffoo



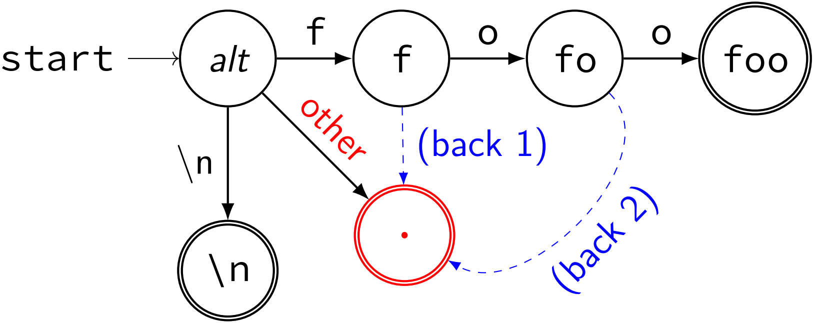
state machine matching

abfoofoabffoo



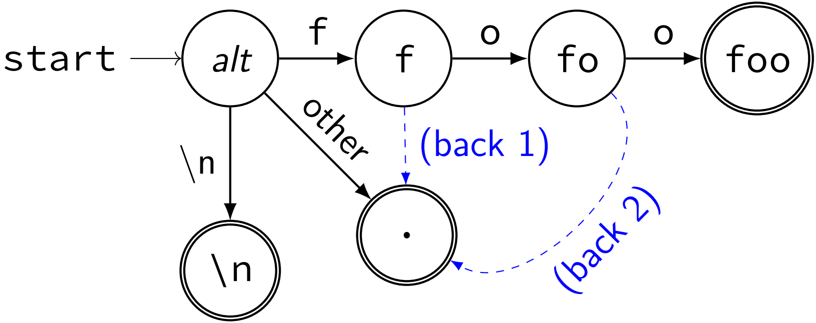
state machine matching

~~a~~bfoofoabffoo



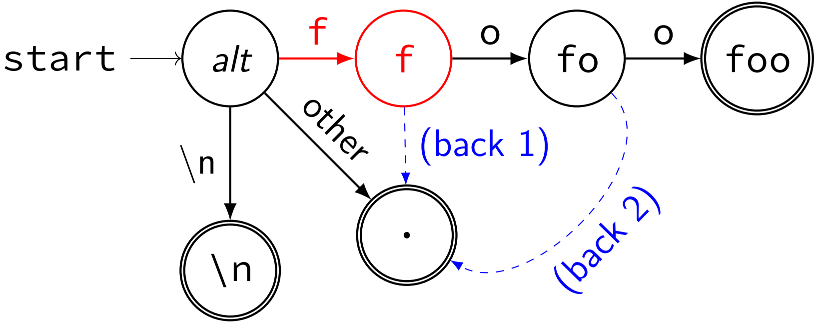
state machine matching

abfoofoabffoo



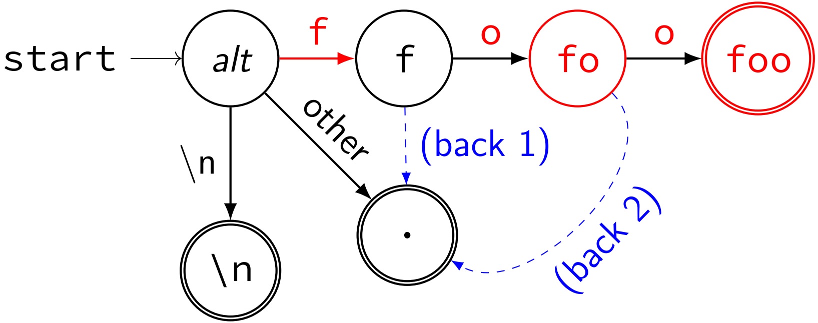
state machine matching

~~ab~~foofoabffoo



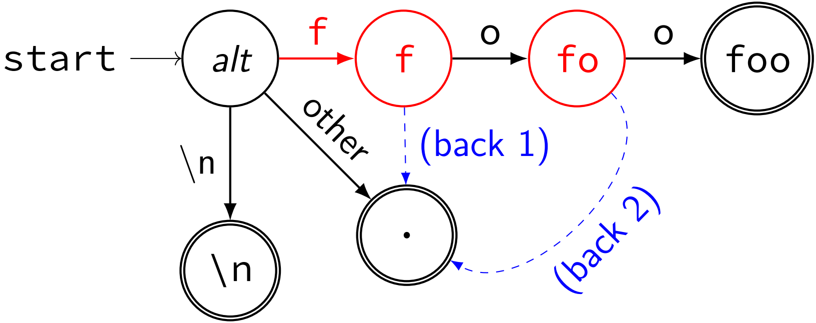
state machine matching

~~ab~~foofoabffoo



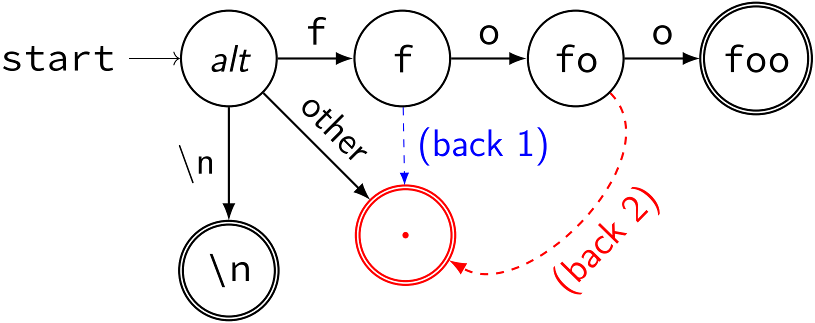
state machine matching

~~abfoo~~foabffoo



state machine matching

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



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

why this?

(basically) one pass matching

basically speed of file I/O

handles multiple patterns well

flexible for “special cases”

why this?

(basically) one pass matching

basically speed of file I/O

handles multiple patterns well

flexible for “special cases”

real anti-virus: probably custom pattern “engine”

other flex features

escape hatch — I/O directly from code

including "unget" function (match normally instead)

allows extra ad-hoc logic

future flex assignment

coming weeks — will have a flex assignment

give you idea what pattern matching can do

produce pattern for `push $…; ret`.

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");
    }
```

avoiding sensitivity: virus patterns

recall: things viruses can't easily change!

example:

- inserted jumps to virus codes
- code in weird parts of executable file
- code that modifies executables
- …

generic generalizing

take static parts of virus

look for distance to match

e.g. `foobarbaz` is 2 from `fooxaxbaz`

slower than regular-expression-like scanners

pattern cost

constructed by hand?

- question: how could we automate?

false positives?

- push + ret really unused?
- jmp at beginning?
- what about data bytes?
- ...

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

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

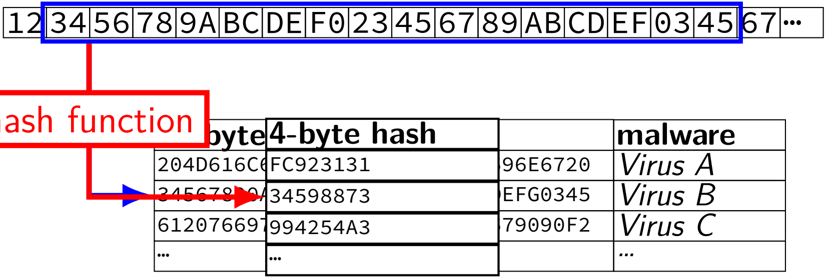
tens of thousands of states???

observation: fixed strings dominate

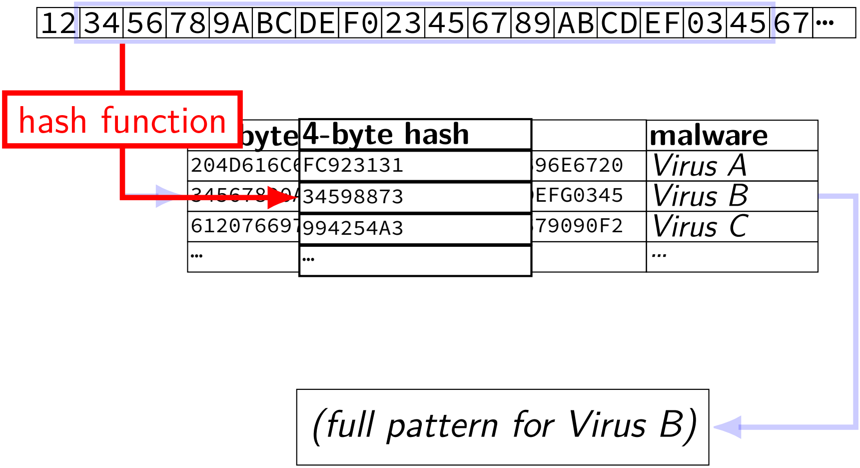
scanning for fixed strings

16-byte "anchor"	**malware**
204D616C6963696F7573205468696E6720	*Virus A*
34567890ABCDEF023456789ABCDEFG0345	*Virus B*
6120766972757320737472696E679090F2	*Virus C*
…	…

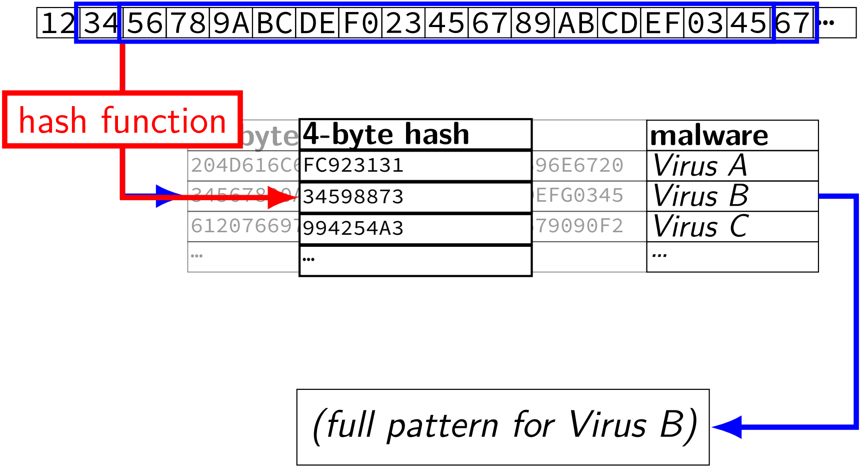
scanning for fixed strings



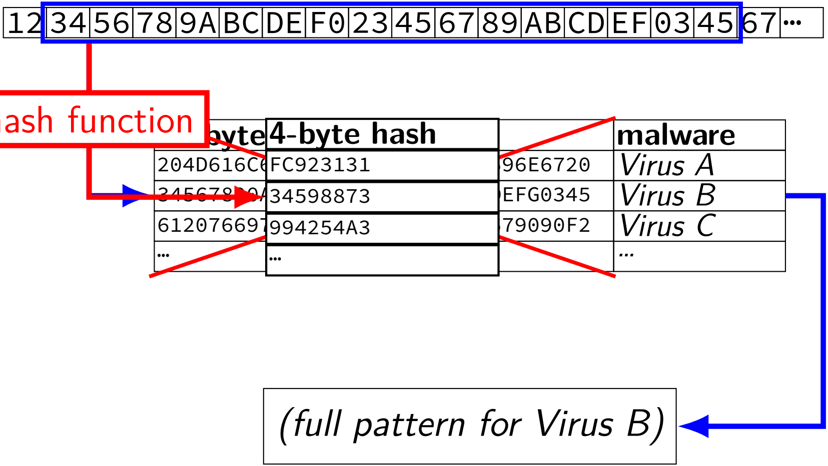
scanning for fixed strings



scanning for fixed strings



scanning for fixed strings



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

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?

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?

head/tail scanning?

playing mouse

techniques so far:

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

virus-writer hat: how can you defeat these?

playing mouse

techniques so far:

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

virus-writer hat: how can you defeat these?

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

playing mouse

techniques so far:

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

virus-writer hat: how can you defeat these?

insert nops everywhere;
split any big strings

playing mouse

techniques so far:

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

virus-writer hat: how can you defeat these?

insert jump in middle
keep code out of end of file

playing mouse: preview

later: metamorphic/polymorphic viruses
- signature resistent
- change every time

anti-analysis techniques
- make reverse engineering harder

playing cat

harder to fool ways of detecting malware?

goal: small changes to malware preserve detection

ideal: detect new malware

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 sectio

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

detecting new malware

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

identify bad behavior

behavior-based detection

things malware does that other programs don't?

basic idea: run in VM; 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 VM; or monitor all programs

anti-virus: essential or worthless?

ungraded homework assignment

watch Hanno Böck's talk "In Search of Evidence-Based IT Security"

a rant mostly about antivirus-like software