anti-anti-virus 2

last time (1)

constructing patterns to match machine code

making pattern matching more efficient

- fast scanning for fixed 'anchor' strings
- head-and-tail scanning — why viruses hide entry points

behavior-based detection

- "hooking" OS calls
- anomalous executable file layout
- apparently looking up library functions by name
- ...

adversarial context — why not standard ML

last time (2)

simple obfuscation transformations

- hide functions by split/merging functions
- hide control flow with switch xform
- general pattern: look to obscure high-level analysis

“encrypted” data

- X ?= Y → hash(X) ?= hash(Y)
- hiding strings from pattern matching with “encryption”
- (not really encryption — key is not secret)

encrypted(?) viruses

```
char encrypted[] = "\x12\x45...";
char key[] = "...";
virusEntryPoint() {
    decrypt(encrypted, key);
    goto encrypted;
}
decrypt(char *buffer, char *key) {...}
```

choose a new key each time!

not good encryption — key is there

sometimes mixed with compression

example: Cascade decrypter

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

Szor Listing 7.1

example: Cascade decrypter

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

Szor Listing 7.1

example: Cascade decrypter

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

Szor Listing 7.1

exercise: some ideas for handling decrypters?

thinking of some anti-decrypter strategies for Cascade

which of the following strategies most practical? least practical?

A. matching patterns of decrypted malware code in memory while executables are running

B. marking executables with too much random-looking data in them

C. matching the decrypter in a normal signature scan

D. trying every possible 'key' for decryption on every executable and matching decrypted malware code against it

E. detecting sequence of file operations Cascade makes instead of its code

decrypter

more variations:

- nested decrypters, different orders, etc.

still problem: decrypter code is signature

...but harder to distinguish different malware

decrypter

more variations:

nested decrypters, different orders, etc.

still problem: decrypter code is signature

...but harder to distinguish different malware

playing mouse

encrypted code? probably still have fast signature from decrypter

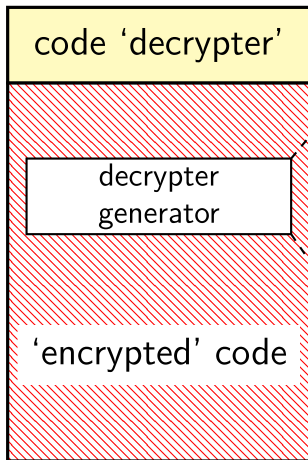
goal: make signatures not work *or* really slow

playing mouse

encrypted code? probably still have fast signature from decrypter

goal: make signatures not work *or* really slow

oligomorphic virus/worm




```
int KEY = RAND();
write(MOV_OPCODE, ...);
...
for (int i = RAND(); i > 0; --i)
    write(NOP_OPCODE);
...
write(XOR_OPCODE, KEY, ...);
...
```

producing changing malware

'encrypted' code can generate new decrypter

not just nop:

switch between synonym instructions

`add $4, ...`, `sub $-4, ...`

swap registers

random instructions that manipulate 'unused' registers

...

template to generate a bunch of decrypters

Szor calls such malware "oligomorphic"

example: W95/Memorial

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

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

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

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

Szor, Listsings 7.3 and 7.4

example: W95/Memorial

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

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

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

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

change instruction order; location of decryption key/etc.

Szor, Listsings 7.3 and 7.4

example: W95/Memorial

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

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

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

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

variable choices of loop instructions

Szor, Listsings 7.3 and 7.4

example: W95/Memorial

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

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

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

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

Szor: “96 different decryptor patterns”

more advanced changes?

Szor calls W95/Memorial oligomoprhic

- “encrypted” code
- plus small changes to decrypter

What about doing more changes to decrypter?

- many, many variations

Szor calls doing this polymorphic

polymorphic example: 1260

example: 1260 (virus)

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

do-nothing instructions

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

different decryption “key”

adapted from Szor, Listing 7.5

‘mutation engine’

```
CopyDecrypter(original_code, new_code) {
    for (each instruction in original_code) {
        new_code += RandomNumberOfNops();
        new_code += PossiblyChooseVariant(instruction)
    }
}
```

terminology: packers

programs that decode and run code at runtime called *packers*

packages exist to do this for non-malware reasons

example motivation: compression

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 scan memory for decrypted machine code

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

using instruction traces (1)

instruction traces are huge...

```
0x10: add %rax, %rbx
0x12: mov 0x140(%rbx), %rsi
0x14: mov %rsi, 0x150(%rbx)
0x16: jle 0x10
0x10: add %rax, %rbx              /* duplicate of before *
0x12: mov 0x140(%rbx), %rsi
0x14: mov %rsi, 0x150(%rbx)
0x16: jle 0x10
0x18: mov $10, %rcx
...
```

but can simplify: e.g. remove duplicates (loops)

using instruction traces (2)

elegant way to analyze 'tricky' techniques

self-modifying code:

```
0x10: add %rax, %rbx
0x12: mov 0x140, %rax
0x14: mov %rsp, 0x0C
    /* modifies code we will execute */
0x16: jle 0x10
0x10: sub %rcx, %rdx
0x12: ...
```

multiple layers of 'decrypters'/code generation

…

stopping packers

it's unusual to jump to code you wrote

modern OSs: memory is executable or writable — not both

stopping packers

it's unusual to jump to code you wrote

modern OSs: memory is executable or writable — not both

diversion: DEP/`W^X`

memory executable or writeable — but not both

exists for exploits (later in course), not packers

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

various names for this feature:

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

special system call to switch modes

unusual, but...

binary translation

- convert machine code to new machine code at runtime

Java virtual machine, JavaScript implementations

- "just-in-time" compilers

dynamic linkers

- load new code from a file — same as writing code?

those packed commercial programs

programs need to explicitly ask for write+exec

exercise: generic detection limits?

consider strategy of running executable in virtual machine,
waiting until it jumps to code it wrote out
then matching patterns against code it's about to run

which of these would cause problems with this technique?

which are easiest/hardest to workaround?

A. code decrypter and malicious code run at program exit, not startup
B. code decrypter and malicious code run when user clicks button in program, not at startup
C. code decrypter allocates random address to write decrypted code to
D. code decrypter exits (without running malicious code) if processor seems too slow
E. code decrypter decrypts another code decrypter

changing bodies

“decrypting” a malware body gives body for “signature”

- “just” need to run decrypter

how about avoiding static signatures entirely

- despite being self-replicating

called metamorphic

- versus polymorphic — only change “decrypter”

example: changing bodies

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

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

code above: after decryption

every instruction changes

still has good signatures

- with alternatives for each possible register selection

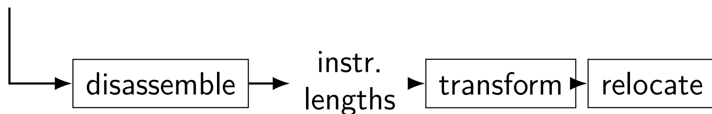
but harder to write/slower to match

case study: Evol

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

"mutation engine"

run as part of propagating the virus



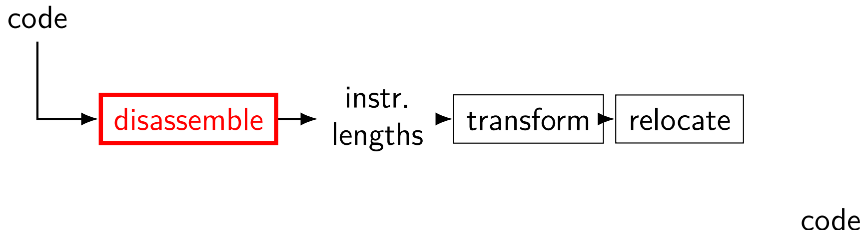
code

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?

virus only handles instructions it has:

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

no prefixes, no floating point

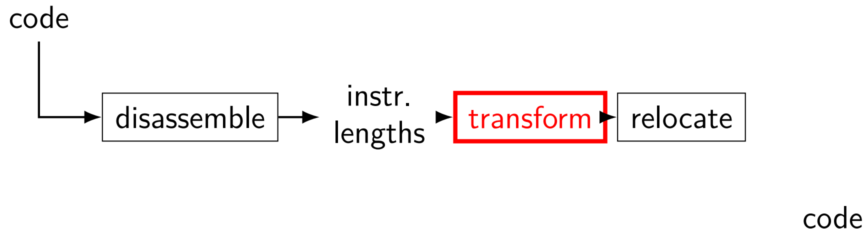
only `%reg` or `$constant` or `offset(%reg)`

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

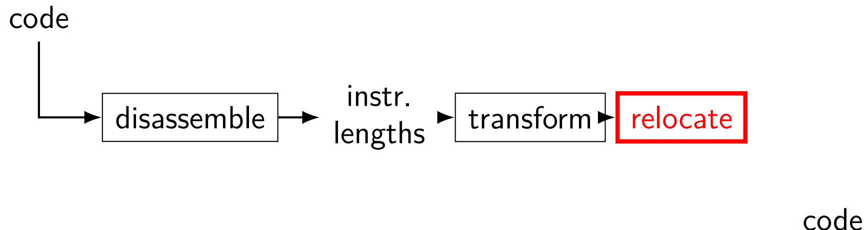
Lakhotia et al., "Are metamorphic viruses really invincible?", Virus Bulletin, Jan 2005

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

problem: mutations mess up jumps/calls

- change were targets of jumps/calls are

table mapping old to new locations

- list of number of bytes generated by each transformation

list of locations references in original

- record relative offset in jump
- record absolute offset in original

relocation example

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

orig. len	new len	instr
5	10	mov1
2	3	mov2
2	7	xor1
1	1	inc1
1	5	dec1
3	3	jne1

address loc	orig. target	new target
$10+3+7+1+5+1$ (jne1+1)	xor1 $(5+2)$	xor1 $(10+3)$

mutation engines

tools for writing polymorphic viruses

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

tedious work to build state-machine-based detector

- ((almost) a regular expression to match it)
- apparently done manually
- automatable?

(but probably can...)

pattern: used until reliably detected

fancier mutation

```
Mutate(original_machine_code, new_machine_code) {
    for (instruction in original_code) {
        new_machine_code += ChooseNewCodeFor(instruction)
    }
    FixupJumpsIn(new_machine_code);
}
```

can do mutation on generic machine code

“just” need full disassembler

identify both instruction lengths and addresses

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

hope to identify tables of function pointers, etc.

fancier mutation

- also an infection technique
  - no “cavity” needed — create one
- obviously tricky to implement
  - need to fix all executable headers
  - what if you misparse assembly?
  - what if you miss a function pointer?
- example: Simile virus

antivirtualization

a lot of malware tries to behave different in a VM

why?

- used by antivirus software to handle packers
- used to analyze malware
- ...

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

- solution: mirror devices of some real machine

time operations that are slower in VM/emulation

use operations not supported by VM

slower operations

not-“native” VM:

- everything is really slow

otherwise — trigger “callbacks” to VM implementation:

- system calls?
- allocating and accessing memory?

...and hope it’s reliably slow enough

antivirtualization techniques

query virtual devices

solution: mirror devices of some real machine

time operations that are slower in VM/emulation

solution: virtual clock

use operations not supported by VM

antivirtualization techniques

query virtual devices

solution: mirror devices of some real machine

time operations that are slower in VM/emulation

solution: virtual clock

use operations not supported by VM

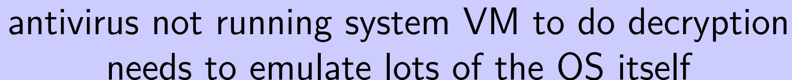
operations not supported

missing instructions kinds?

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

not handling OS features?

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



attacking emulation patience

looking for unpacked virus in VM

...or other malicious activity

when are you done looking?

attacking emulation patience

looking for unpacked virus in VM

…or other malicious activity

when are you done looking?

malware solution: take too long

- not hard if emulator uses "slow" implementation

attacking emulation patience

looking for unpacked virus in VM

...or other malicious activity

when are you done looking?

malware solution: take too long

- not hard if emulator uses "slow" implementation

malware solution: don't infect consistently

probability

```
if (randomNumber() == 4) {
    unpackAndRunEvilCode();
}
```

antivirus emulator:

`randomNumber() == 3`

looks clean!

real execution #1:

`randomNumber() == 2`

no infection!

real execution $\#N$:

`randomNumber() == 4`

infect!

attacking emulation patience

looking for unpacked virus in VM

…or other malicious activity

when are you done looking?

malware solution: take too long

- not hard if emulator uses "slow" implementation

malware solution: don't infect consistently

malware solution: use more memory, etc.

on goats

analysis and maybe detection uses *goat files*

"sacrificial goat" to get changed by malware

heuristics can avoid simple goat files, e.g.:

- don't infect small programs
- don't infect huge programs
- don't infect programs with huge amounts of nops
- ...

backup slides

encrypted viruses: no signature?

`decrypt` is a pretty good signature

still need to a way to disguise that code

how about analysis? how does one analyze this?

encrypted virus: getting the code?

"encrypted" body

just running objdump not enough...

instead — run debugger, set breakpoint after "decryption"

dump decrypted memory afterwords

observation: can even automate this:

- run program in emulator
- have emulator look for jump to previously written code
- (or jump after certain point, etc.)
- example implementation: unipacker (`https://github.com/unipacker/unipacker`)