# anti-anti-virus

defeating signatures:

avoid things compilers/linkers never do

make analysis harder
- takes longer to produce signatures
- takes longer to produce “repair” program
- may evade attempts to automate analysis

make changing viruses
- make signatures less effective

# some terms

*armored* viruses

viruses designed to make analysis harder

*metamorphic*/*polymorphic*/*oligomorphic* viruses

viruses that change their code each time
different terms — different types of changes (later)

# obfuscation, generally

malware often *obfuscates* (obscures) its code

several reasons for this

- prevent their from being signatures
- make analysis more difficult
- prevent others from modifying+copying

note: many of these technique sometimes employed by commercial software

- intended to prevent copying/reverse-engineering

# Tigress as example of obfuscation

Tigress — researcher developer obfuscation tool

`https://tigress.wtf`

includes many *transformations* typical of real-world obfuscation

we'll talk about the ideas behind many of them

future assignment involving code obfuscated with Tigress

# example Tigress transformations

we'll look at some simple ones Tigress provides

I'm showing you the pattern,
not the actual code Tigress generates

# Tigress: provided transform: EncodeLiterals

```
void Print() { printf("Hello!"); printf("World!"); }
```

---

```
void GetString(int x, char *buffer) {
    switch (x) {
    case 0:
        buffer[0] = 'H'; buffer[1] = 'e'; buffer[2] = 'l'; ...;
        break;
    case 1:
        buffer[0] = 'W'; buffer[1] = 'o'; buffer[2] = 'r'; ...;
        break;
    case 2:
    ...
    }
}
void Print() { char buf[100];
    GetString(0, buf); printf(buf);
    GetString(1, buf); printf(buf);
}
```

# Tigress: provided transform: Merge

```
void foo(int a) { code for foo }
void bar(int a) { code for bar }

... foo(x) + bar(y) ...
```

---

```
void foo_bar(int s, int a) {
    if (s == 0) {
        code for foo
    } else {
        code for bar
    }
}

... foo_bar(0, x) + foo_bar(1, y) ...
```

# Tigress: provided transform: Split

```
void foo(int a, int b) {
    int x = ...;
    code for foo part 1
    code for foo part 2
}
```

---

```
void foo1(int *a, int *b, int *x) {
    code for foo part 1
}
void foo2(int *a, int *b, int *x) {
    code for foo part 2
}
void foo(int a, int b) {
    int x;
    foo1(&a,&b,&x); foo2(&a,&b,&x);
}
```

# Tigress: example transform: Flatten

```
void foo() {
    A;
    if (X) {
        B;
    } else {
        C;
    }
    D;
}
```

---

```
void foo() {
    int s = 0;
    while (1) {
        switch(s) {
        case 0: A; s = X ? 1 : 2; break;
        case 1: B; s = 3; break;
        case 2: C; s = 3; break;
        case 3: D; return;
        }
    }
```

# some other xforms

add useless conditionals, etc.

encode constants like strings

generate machine code at runtime and jump to it

turn function into custom bytecode interpreter

- data array holding effective function code

add checks that machine code hasn't changed

# transformations so far?

all can be combined!

annoying for analysis

hard to do without unobfuscated code

- can't easily be redone/changed by self-replicating malware

probably more distinctive than original code for signatures

- (just match the transformed version since it won't change often)

next topic: transformations to avoid signatures

- (Tigress supports those, but not our primary examples)

# obfuscation versus analysis

which of these does obfuscation seem most/least likely to hamper doing?

A. determining what remote servers some malware contacts

B. determining a password the malware requires to access extra functionality

C. accessing extra functionality in the malware protected by a password

D. determining whether the malware will behave differently based on the time

# recall: library calls in viruses

viruses making library calls

- can't use normal dynamic linker stuff

common solution: search by name:

```
char *names[] = GetFunctionNamesFrom("kernel32.dll");
for (int i = 0; i < numFunctions; ++i) {
    if (strcmp(names[i], "GetFileAttributesA") == 0) {
        return functions[i];
    }
}
```

problem: legit application code won't do this

easy to look for string 'GetFileAttributesA'

# searching for hashes

```
char *functionNames[] = GetFunctionsFromStandardLibrary();
/* 0xd7c9e758 = hash("GetFileAttributesA") */
unsigned hashOfString = 0xd7c9e758;
for (int i = 0; i < num_functions; ++i) {
    unsigned functionHash = 0;
    for (int j = 0; j < strlen(functionNames[i]); ++j) {
        functionHash = (functionHash * 7 +
                        functionNames[i][j]);
    }
    if (functionHash == hashOfString) {
        return functions[i];
    }
}
```

# encrypted(?) data

```
char obviousString[] =
    "Please␣open␣this␣100%"
    "␣safe␣attachment";
char lessObviousString[] =
    "oSZ^LZ\037POZQ\037KWVL\037\016\017"
    "\017\032\037L^YZ\037^KK^\\WRZQK";
for (int i = 0; i < sizeof(lessObviousString) - 1; ++i) {
    lessObviousString[i] =
        lessObviousString[i] ^ '?';
}
```

# encrypted data and signatures

get rid of some easy signatures

- especially if ‘key’ changes or hashes used

but not enough:

- decryption code is very distinctive

# encrypted data and signatures

get rid of some easy signatures

- especially if ‘key’ changes or hashes used

but not enough:

- decryption code is very distinctive

can we do better with this “encryption” idea?

# 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:
    ...
```

# 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

# 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*

"disinfection" — want to precisely identify malware

# playing mouse

encrypted code? probably still have fast signature from decrypter

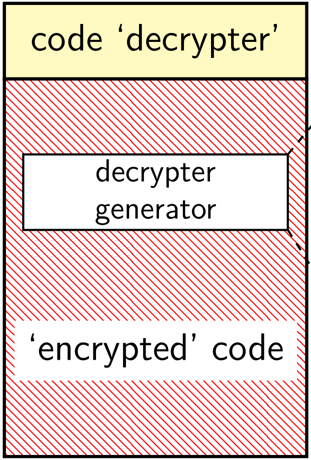
goal: make signatures not work *or* really slow

# 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

decry
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

nop
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

  nop
  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

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

Szor: “96 different decryptor patterns”

Szor, Listsings 7.3 and 7.4

# 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
decrypt:
    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
decrypt:
    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*

packers exist to do this for non-malware reasons

example motivations:

- compression
- packaging libraries + executable together

# from UPX documentation

## Introduction

**UPX** is an advanced executable file compressor. UPX will typically reduce the file size of programs and DLLs by around 50%-70%, thus reducing disk space, network load times, download times and other distribution and storage costs.

Programs and libraries compressed by **UPX** are completely self-contained and run exactly as before, with no runtime or memory penalty for most of the supported formats.

---

```
 - We will *NOT* add any sort of protection and/or encryption.
   This only gives people a false feeling of security because
   all "protectors" can be broken by definition.
```

# 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

# unicorn as tool

## Unicorn

The Ultimate CPU emulator

Service Download Docs Showcase Contact

**Unicorn** is a lightweight multi-platform, multi-architecture CPU emulator framework.

Highlight features:

- Multi-architectures: ARM, ARM64 (ARMv8), m68k, MIPS, PowerPC, RISC-V, S390x (SystemZ), SPARC, TriCore & x86 (include x86_64).
- Clean/simple/lightweight/intuitive architecture-neutral API.
- Implemented in pure C language, with bindings for Pharo, Crystal, Clojure, Visual Basic, Perl, Rust, Haskell, Ruby, Python, Java, Go, D, Lua, JavaScript, .NET, Delphi/Pascal & MSVC available.
- Native support for Windows & *nix (with macOS, Linux, Android, *BSD & Solaris confirmed).
- High performance by using Just-In-Time compiler technique.
- Support fine-grained instrumentation at various levels.

# unicorn example (1)

```
$ cat test.s
    mov $10000, %edi
    imul $2, %rdi, %rdi
$ gcc -c test.s; objcopy -j .text test.o -O binary test.bin
```

```
code = Path('test.bin').read_bytes()
uc = Uc(UC_ARCH_X86, UC_MODE_64)
uc.mem_map(0x10000, 1024 * 1024)
uc.mem_write(0x10000, code)
uc.emu_start(0x10000, 0x10000 + len(code))
print("RDI",uc.reg_read(UC_X86_REG_RDI))
```

```
RDI 20000
```

# unicorn example (2)

```
...
uc.hook_add(UC_HOOK_CODE, hook_code_func)
def hook_code_func(uc, addr, size, user_data):
    print(f"{addr:x} ({size} byte instruction): "
          f"{codecs.encode(
                uc.mem_read(addr, size), 'hex'
             ).decode()}")
uc.emu_start(0x10000, 0x10000 + len(code))
```

---

```
10000 (5 byte instruction): bf10270000
10005 (4 byte instruction): 486bff02
```

# unipacker psuedocode

```
data, size = parse_executable()
uc.mem_map(BASE_ADDR, size)
uc.mem_write(BASE_ADDR, data)
for dll in get_executable_libraries():
    uc.mem_map(dll['addr'], dll['size'])
    uc.mem_write(dll['addr'], dll['data'])
uc.hook_add(UC_HOOK_CODE, before_execute)
...
uc.emu_start(...)

# called before each instruction
def before_execute(uc, addr, ...):
    if in_modified_section(addr):
        dump_memory_now()
    ...
```

# example tool: qiling

https://qiling.io

uses Unicorn emulator but adds...

emulation for a lot of system calls

- including (hopefully) limiting file accesses to specific “virtual root” directory

loaders for common executable/bootloader formats

idea: get log of malware activity / add custom behaviors

# PANDA.re

fork of emulator QEMU

supports whole-system record+replay

idea: run virtual machine with malware

replay run with analyses that can look at all instructions run

examples:

- identify where dat from a specific file was used
- search memory for string throughout execution
- function call history

# traces instead of unpacked code

instead of matching signatures on code at rest

can match signature on *trace* of executed instructions

# 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

...

# 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
- ...

antivirus not running system VM to do decryption needs to emulate lots of the OS itself

# 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.*

# stopping packers

it's unusual to jump to code you wrote

modern OSs/compilers: memory not writeable and executable

# stopping packers

it's *unusual* to jump to code you wrote

modern OSs/compilers: memory not writeable and executable

```
LOAD off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**12
     filesz 0x00003458 memsz 0x00003458 flags r--
LOAD off    0x00004000 vaddr 0x00004000 paddr 0x00004000 align 2**12
     filesz 0x00013091 memsz 0x00013091 flags r-x
LOAD off    0x00018000 vaddr 0x00018000 paddr 0x00018000 align 2**12
     filesz 0x00007458 memsz 0x00007458 flags r--
LOAD off    0x0001ffd0 vaddr 0x00020fd0 paddr 0x00020fd0 align 2**12
     filesz 0x000012a8 memsz 0x00002570 flags rw-
```

# 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)

usually *special system call* to switch modes

- Linux: `mprotect`

# 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

# OBFUSCATE assignment

modifying ‘password-protected’ tic-tac-toe game not to be

three versions:

- just stripped
- some Tigress transformations
- ‘encrypted’ code

# 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

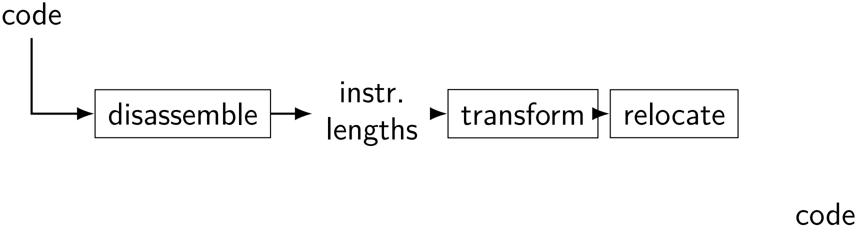
but harder to write/slower to match

# 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

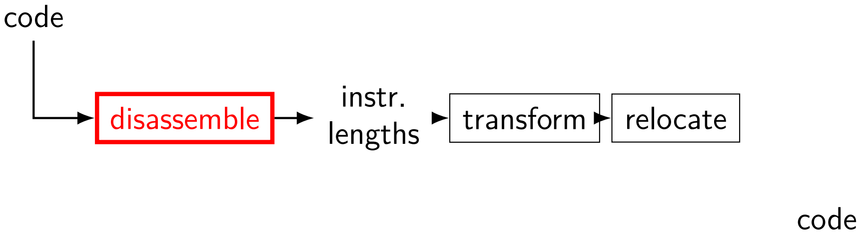


# case study: Evol

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

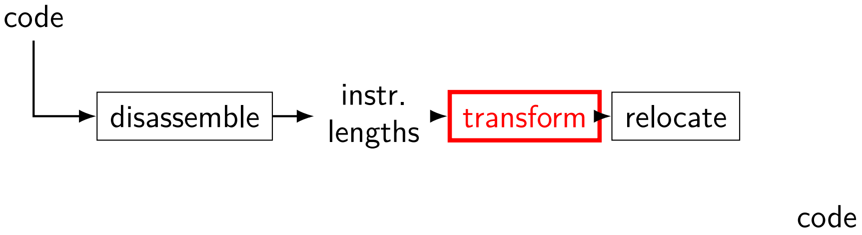
only `%reg` or `$constant` or `offset(%reg)`

# 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

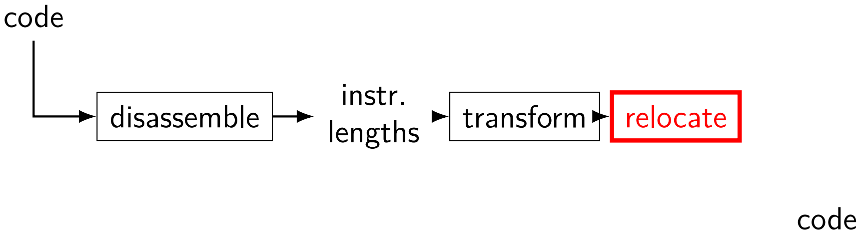
Lakhotia et al., "Are metamorphic viruses really invincible?", Virus Bulletin, Jan 2005

# 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*

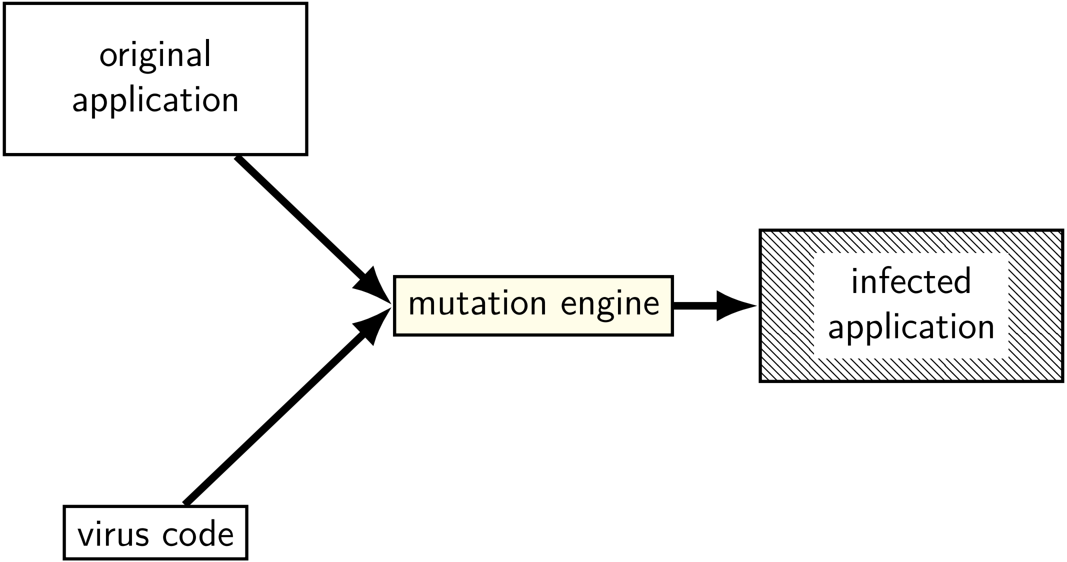
“just” need full disassembler

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.

# mutation as infection technique



# fancier mutation

no "cavity" needed — create one

- insert virus code by adjusting surrounding cod

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

# 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`
- …

# 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`

*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

- exception handler = way for CPU to run OS code
- (despite no actual normal jmp/etc. to 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:
    ...
        /* RBX = current sum; RAX = pointer to code */
    movq $0, %rbx              // Intel: mov RBX, 0
    movq $mycode, %rax         // Intel: mov RAX, OFFSET MYCODE
loop:
    addq (%rax), %rbx          // Intel: add RBX, [RAX]
    addq $8, %rax              // Intel: add 8, RAX
        /* current sum += *code_ptr;  code_ptr += ... */
    cmpq $endcode, %rax
    jl loop
    cmpq %rbx, $EXPECTED_VALUE
    jne debugger_found      // if sum wrong, panic
    ...
```

# 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? probably fallback to `int 3`

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

# anti-single-step

x86: single-stepping implemented with processor flag

- causes OS to run after every instruction

can read flag normally with common debugger configurations

- more modern systems may support hiding better

could also check timing

could also try to replace OS's single-step handler

# emulation based obfuscation

so far: always producing machine code and running it

analyzing machine code with virtual machine, debugger, etc.

alternate idea: invent a new instruction set

convert program to that instruction set

include interpreter for that instruction set

# example: Tigress Virtualize transform (1)

input:

```
int example(int x) {
    if (x > 10) {
        printf("Yes!\n");
    }
}
```

Tigress generates instruction set for stack-based machine

- uses little stack instad of registers for most instructions
- same design used by, e.g., Java VM

instructions can pop+push from stack for temporaries

# example: Tigress Virtualize transform (2)

instruction set for example

- call OPERAND=funcId with arguments LOCALS[1]
- pop t1, pop t2, push t1>t2
- push OPERAND
- push table[OPERAND]
  - different variants for int, string, ...
- pop t1, LOCALS[OPERAND] = t1
- pop t1, if (t1) goto OPERAND
- return

customized for this function

each instruction has opcode, variable length (if operands)

# example: Tigress Virtualize transform (3)

```
int example(int x) {
    if (x > 10) {
        printf("Yes!\n");
    }
}
```

each line below one "instruction"

- (actually encoded as part of array of bytes)
- push OPERAND=10
- push table[OPERAND=...] (argument x)
- pop t1 pop t2 push t1>t2
- pop t1, if (t1) goto OPERAND=OUT
- push table[OPERAND=...] (string `"Yes!"`)
- pop t1, LOCALS[OPERAND=1] = t1
- call OPERAND=...(printf) with arguments LOCAL1
- OUT: ...

# example: Tigress emulator

```
_1_example_$sp[0] = _1_example_$stack[0];
_1_example_$pc[0] = _1_example_$array[0];
while (1) {
  switch (*(_1_example_$pc[0])) {
  ...
  }
}
```

pc variable representing emulated stack

- switch statement based on opcode

sp variable representing emulated stack

# effectiveness of this transformation?

huge performance impact

can do analysis on new instruction set

- how much more difficult than working with original machine code?

instruction traces still helpful

- about as easy to get record of everything done

# attacking antivirus (1)

one common virus idea: interfere directly with antivirus

just modify antivirus software databases, etc.

preserve file checksums

- so some AV software thinks file is unchanged
- (doesn't work with cryptographic hashes, but…)

register own handlers to filter antivirus/sysadmin calls

# attacking antivirus (1)

one common virus idea: interfere directly with antivirus

just modify antivirus software databases, etc.

preserve file checksums

so some AV software thinks file is unchanged
(doesn't work with cryptographic hashes, but...)

*register own handlers to filter antivirus/sysadmin calls*

# stealth

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

# other 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”

# 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

# 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.

# backup slides

# 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

ars TECHNICA

AI BIZ & IT CARS CULTURE GAMING HEALTH POLICY SCIENCE SECURITY SPACE TECH FORUM | SUBSCRIBE

MORE SECURITY THEATER

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

# 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`)