antiantivirus / buffer overflows

Changelog

4 Mar 2021: add stack layout exercise answers; metamorphic section on last time slides

last time (1)

"encrypted"/packed code pattern

- "decrypter" unpacks code + jumps to it

oligomorphic malware

- hide decrypter from pattern matching with multiple variants
- generally: template with few blanks

polymorphic malware

- generate/mutate decrypter in more generic way
- "decrypted" code still unchanged

last time (2)

handling packers in antimalware software

- generic algorithm: run in emulator/VM, then
- examine memory for unpacked code at appropriate time and/or
- record list of instructions executed

metamorphic malware

- apply "mutation engine" to change entire machine code
- avoids leaving signatures in memory after "decrypter" runs

antivirtualization/emulation

- query machine devices
- unimplemented instructions, features

logistical note

LOCATION in LEX: offset in file, please

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
    ...
endcode:
```

detecting int 3 directly (2)

query the "handler" for int 3

- old OSs only; today: cannot set directly

modern OSs: ask if there's a debugger attached

...or try to attach as debugger yourself

- doesn't work — debugger present, probably
- does work — broke any debugger?

```
 // Windows API function!
if (IsDebuggerPresent()) { ... }
```

modern debuggers

`int 3` is the oldest x86 debugging mechanism

modern x86: 4 "breakpoint" registers (DR0–DR3)

- contain address of program instructions
- need more than 4? sorry
  - probably fall back to int 3 technique

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

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.

vulnerabilities

for viruses, worms

for trojans + PUP that do more than is supposed to do be allowed

e.g. getting location information without "permission"

software vulnerability

unintended program behavior
that can be used by an adversary

vulnerability example

website able to install software without prompting

not intended behavior of web browser

software vulnerability classes (1)

memory safety bugs

- problems with pointers
- big topic in this course

"injection" bugs — type confusion

- commands/SQL within name, label, etc.

integer overflow/underflow

...

software vulnerability classes (2)

not checking inputs/permissions

```
http://webserver.com/../../../../file-I-shouldn't-get.txt
```

almost any 's "undefined behavior" in C/C++

synchronization bugs: time-to-check to time-of-use

... more?

vulnerability versus exploit

exploit — something that uses a vulnerability to do something

proof-of-concept — something = demonstration the exploit is there

example: open a calculator program

typical buffer overflow pattern

cause program to write past the end of a buffer

that somehow causes different code to run

(usually code the attacker wrote)

why buffer overflows?

for a long time, most common vulnerability

common results in arbitrary code execution

related to other memory-management vulnerabilities

- which usually also result in arbitrary code execution

network worms and overflows

worms that connect to vulnerable servers:

Morris worm included some buffer overflow exploits

> Morris worm: first self-replicating malware
> in mail servers, user info servers

2001: Code Red worm that spread to web servers (running Microsoft IIS)

overflows without servers

bugs dealing with corrupt files:

Adobe Flash (web browser plugin)

PDF readers

web browser JavaScript engines

image viewers

movie viewers

decompression programs

...

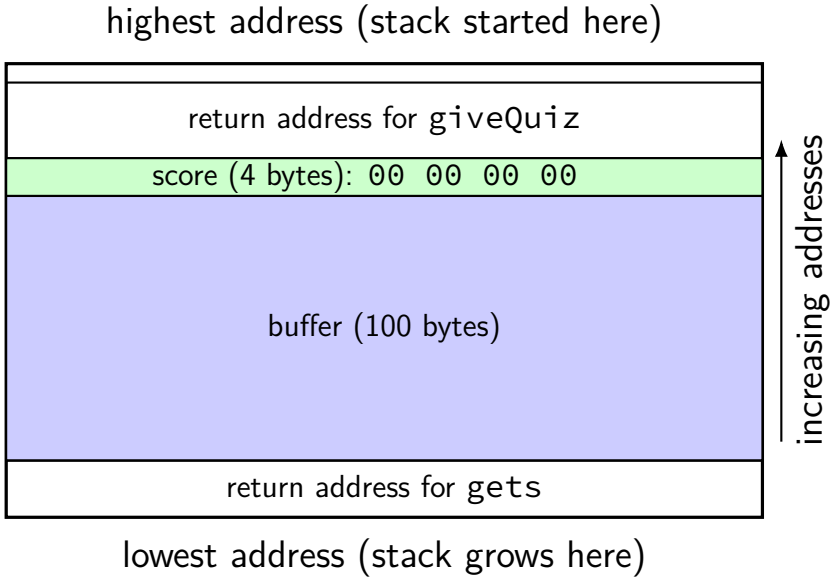
simpler overflow

```
struct QuizQuestion questions[NUM_QUESTIONS];
int giveQuiz() {
    int score = 0;
    char buffer[100];
    for (int i = 0; i < NUM_QUESTIONS; ++i) {
        gets(buffer);
        if (checkAnswer(buffer, &questions[i])) {
            score += 1;
        }
    }
    return score;
}
```

simpler overflow

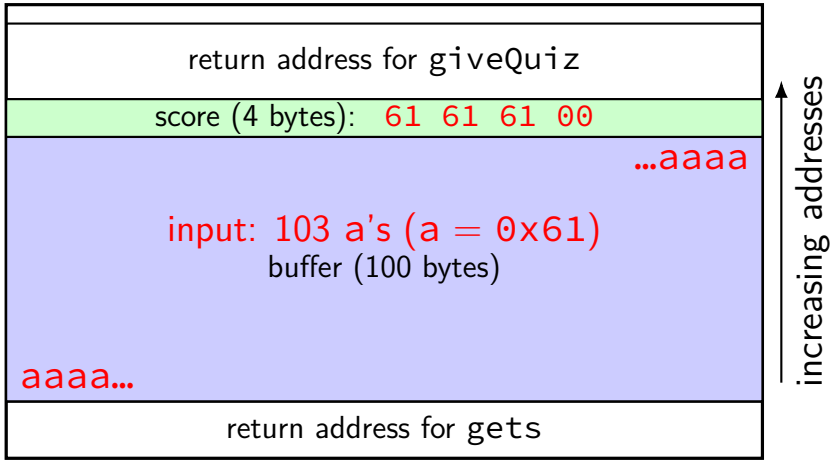
```
struct QuizQuestion questions[NUM_QUESTIONS];
int giveQuiz() {
    int score = 0;
    char buffer[100];
    for (int i = 0; i < NUM_QUESTIONS; ++i) {
        gets(buffer);
        if (checkAnswer(buffer, &questions[i])) {
            score += 1;
        }
    }
    return score;
}
```

simpler overflow: stack



simpler overflow: stack

highest address (stack started here)



lowest address (stack grows here)

Stack Smashing

original, most common buffer overflow exploit

worked for most buffers on the stack

("worked"? we'll talk later)

Aleph1, Smashing the Stack for Fun and Profit

"non-traditional literature"; released 1996

by Aleph1 AKA Elias Levy

.oO Phrack 49 Oo.

Volume Seven, Issue Forty-Nine

File 14 of 16

BugTraq, r00t, and Underground.Org
bring you

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Smashing The Stack For Fun And Profit
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

by Aleph One
aleph1@underground.org

exercise: stack layout

```
GradeAssignment:
  pushq   %rbp
  pushq   %rbx
  xorl    %ebx, %ebx
  subq    $72, %rsp
  leaq    8(%rsp), %rbp
for_loop:
  movq    %rbp, %rdi
  call    gets
  movl    %ebx, %esi
  movq    %rbp, %rdi
  call    GradeAnswer
  leaq    24(%rsp), %rdi
  movl    %eax, (%rdi,%rbx,4)
  incq    %rbx
  cmpq    $10, %rbx
  jne     for_loop
  call    Process
  ...
```

```
int GradeAssignment(FILE *in) {
  int scores[10]; char buffer[16];
  for (int i = 0; i < 10; ++i) {
    gets(buffer);
    scores[i] =
        GradeAnswer(buffer, i);
  }
  Process(scores);
}
```

exercise: how many bytes after `buffer[0]` is the first byte of `scores[0]`?

exercise: stack layout

```
GradeAssignment:
  pushq   %rbp
  pushq   %rbx
  xorl    %ebx, %ebx
  subq    $72, %rsp
  leaq    8(%rsp), %rbp
for_loop:
  movq    %rbp, %rdi
  call    gets
  movl    %ebx, %esi
  movq    %rbp, %rdi
  call    GradeAnswer
  leaq    24(%rsp), %rdi
  movl    %eax, (%rdi,%rbx,4)
  incq    %rbx
  cmpq    $10, %rbx
  jne     for_loop
  call    Process
  ...
```

```
int GradeAssignment(FILE *in) {
  int scores[10]; char buffer[16];
  for (int i = 0; i < 10; ++i) {
    gets(buffer);
    scores[i] =
        GradeAnswer(buffer, i);
  }
  Process(scores);
}
```

exercise: how many bytes after `buffer[0]` is the first byte of `scores[0]`? answer: 16

exercise: overflow?

```
GradeAssignment:
  pushq   %rbp
  pushq   %rbx
  xorl    %ebx, %ebx
  subq    $72, %rsp
  leaq    8(%rsp), %rbp
for_loop:
  movq    %rbp, %rdi
  call    gets
  movl    %ebx, %esi
  movq    %rbp, %rdi
  call    GradeAnswer
  leaq    24(%rsp), %rdi
  movl    %eax, (%rdi,%rbx,4)
  incq    %rbx
  cmpq    $10, %rbx
  jne     for_loop
  call    Process
  ...
```

```
int GradeAssignment(FILE *in) {
  int scores[10]; char buffer[16];
  for (int i = 0; i < 10; ++i) {
    gets(buffer);
    scores[i] =
        GradeAnswer(buffer, i);
  }
  Process(scores);
}
```

exercise: if input into buffer is 50 copies of the character `'1'` what is value of `scores[0]`?

exercise: overflow?

```
GradeAssignment:
  pushq   %rbp
  pushq   %rbx
  xorl    %ebx, %ebx
  subq    $72, %rsp
  leaq    8(%rsp), %rbp
for_loop:
  movq    %rbp, %rdi
  call    gets
  movl    %ebx, %esi
  movq    %rbp, %rdi
  call    GradeAnswer
  leaq    24(%rsp), %rdi
  movl    %eax, (%rdi,%rbx,4)
  incq    %rbx
  cmpq    $10, %rbx
  jne     for_loop
  call    Process
  ...
```

```
int GradeAssignment(FILE *in) {
  int scores[10]; char buffer[16];
  for (int i = 0; i < 10; ++i) {
    gets(buffer);
    scores[i] =
        GradeAnswer(buffer, i);
  }
  Process(scores);
}
```

exercise: if input into buffer is
50 copies of the character `'1'`
what is value of `scores[0]`? answer:

backup slides

unstealthy debuggers

is a debugger installed?

- unlikely on Windows, maybe ignore those machines

is a debugger process running (don't check if it's tracing you)

...

confusing debuggers

"broken" executable formats

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

use the stack pointer not for the stack

- stack trace?

recall: virus code

```
    leal string(%rip), %edi
    pushq $0x4004e0 /* address of puts */
    retq
string:
    .asciz "You have been infected with a virus!"
```

recall: virus code

```
    leal string(%rip), %edi
    pushq $0x4004e0 /* address of puts */
    retq
string:
    .asciz "You have been infected with a virus!"
```

```
8d 3d 06 00 00 00 (leal)
```

opcode for lea
ModRM byte:
    32-bit displacement; %rdi
32-bit offset from instruction

recall: virus code

```
    leal string(%rip), %edi
    pushq $0x4004e0 /* address of puts */
    retq
string:
    .asciz "You have been infected with a virus!"
```

```
8d 3d 06 00 00 00 (leal)
68 e0 04 40 00 (pushq)
```

opcode for push 32-bit constant
32-bit constant (extended to 64-bits)

recall: virus code

```
    leal string(%rip), %edi
    pushq $0x4004e0 /* address of puts */
    retq
string:
    .asciz "You have been infected with a virus!"
```

```
8d 3d 06 00 00 00 (leal)
68 e0 04 40 00 (pushq)
c3 (retq)
```

virus code to shell-code (1)

```
    leaq string(%rip), %rdi
    pushq $0x4004e0 /* address of puts */
    retq
string:
    .asciz "You have been infected with a virus!"
```

```
48 8d 3d 06 00 00 00 (leaq)
68 e0 04 40 00 (pushq)
c3 (retq)
```

REX prefix for 64-bit
opcode for lea
ModRM byte: 32-bit displacement; %rd
32-bit offset from instruction

virus code to shell-code (1)

```
    leaq string(%rip)
    pushq $0x4004e0 /
    retq
string:
    .asciz "You have been infected with a virus!"
```

leaq not leal
stack address > 0xFFFF FFFF

```
48 8d 3d 06 00 00 00 (leaq)
68 e0 04 40 00 (pushq)
c3 (retq)
```

REX prefix for 64-bit
opcode for lea
ModRM byte: 32-bit displacement; %rd
32-bit offset from instruction

virus code to shell-code (1)

```
    leaq string(%rip),
    pushq $0x4004e0 /*
    retq
string:
    .asciz "You have been infected with a virus!"
```

problem: what if we don't know where puts is?

```
48 8d 3d 06 00 00 00 (leaq)
68 e0 04 40 00 (pushq)
c3 (retq)
```

REX prefix for 64-bit
opcode for lea
ModRM byte: 32-bit displacement; %rd
32-bit offset from instruction

virus code to shell-code (2)

```
    /* Linux system call (OS request):
       write(1, string, length)
     */
    leaq string(%rip), %rsi
    movl $1, %eax
    movl $37, %edi
    /* "request to OS" instruction */
    syscall
string:
    .asciz "You have been infected with a virus!\n"
```

```
48 8d 35 0c 00 00 00 (leaq)
b8 01 00 00 00 (movq %eax)
bf 25 00 00 00 (movq %edi)
0f 05 (syscall)
```

virus code to shell-code (2)

```
    /* Linux system call (OS request):
       write(1, string, length)
     */
    leaq string(%rip), %rsi
    movl $1, %eax
    movl $37, %edi
    /* "request to OS" instruction */
    syscall
string:
    .asciz "You have been infected with a virus!\n"
```

```
48 8d 35 0c 00 00 00 (leaq)
b8 01 00 00 00 (movq %eax)
bf 25 00 00 00 (movq %edi)
0f 05 (syscall)
```

problem: after syscall — crash!

virus code to shell-code (3)

```
    /* Linux system call (OS request):
       write(1, string, length)
     */
    leaq string(%rip), %rsi
    movl $1, %eax
    movl $37, %edi
    syscall
    /* Linux system call:
       exit_group(0)
     */
    movl $231, %eax
    xor %edi, %edi
    syscall
string:
    .asciz "You have been infected with a virus!\n"
```

virus code to shell-code (3)

tell OS to exit

```
    /* Linux system call (OS request):
       write(1, string, length)
     */
    leaq string(%rip), %rsi
    movl $1, %eax
    movl $37, %edi
    syscall
    /* Linux system call:
       exit_group(0)
     */
    movl $231, %eax
    xor %edi, %edi
    syscall
string:
    .asciz "You have been infected with a virus!\n"
```

diversion: debuggers

we'll care about two pieces of functionality:

breakpoints
: debugger gets control when certain code is reached

single-step
: debugger gets control after a single instruction runs

implementing single-stepping (1)

set a breakpoint on the following instruction?

```
movq %rax, %rdx
addq %rbx, %rdx // <-- STOPPED HERE
subq 0(%rsp), %r8 // <-- SINGLE STEP TO HERE
subq 8(%rsp), %r8
...
```

transformed to

```
movq %rax, %rdx
addq %rbx, %rdx // <-- STOPPED HERE
int 3 // <-- SINGLE STEP TO HERE
subq 8(%rsp), %
...
```

then `jmp` to `addq`

implementing single-stepping (1)

set a breakpoint on the following instruction?

```
movq %rax, %rdx
addq %rbx, %rdx // <-- STOPPED HERE
subq 0(%rsp), %r8 // <-- SINGLE STEP TO HERE
subq 8(%rsp), %r8
...
```

transformed to

```
movq %rax, %rdx
addq %rbx, %rdx // <-- STOPPED HERE
int 3 // <-- SINGLE STEP TO HERE
subq 8(%rsp), %
...
```

then `jmp` to `addq`

but what about

```
jmpq *0x1234(%rax,%rbx,8) // <-- STOPPED HERE
```

implementing single-stepping (2)

typically hardware support for single stepping

x86: `int 1` handler (second entry in table)

x86: `TF` flag: execute handler after every instruction

...except during handler (whew!)

Defeating single-stepping

try to install your own `int 1` handler

(if OS allows)

try to clear TF?

would take effect on following instruction

...if debugger doesn't reset it