Exam Review

logistical note

post-exam stack smashing assignment

due two weeks after spring break (was one on schedule, but...)

likely harder than tricky — will count for more

exam format

around 20 question parts

mostly multiple choice or multiple-multiple choice

something similar to RE

something similar to TRICKY

something about antiantivirus strategies, VMs, etc.

given information

X86-64 calling convention reminder:

- first argument: `%rdi`
- second argument: `%rsi`
- return value: `%rax`
- return address: on stack

X86-64 registers reminder:

- `%rax` (64-bit), `%eax` (lower 32 bits), `%ax` (lower 16 bits), `%al` (lower 8 bits)
- (and similar for `%rbx`, `%rcx`, `%rdx`)
- `%rsi` (64-bit), `%esi` (lower 32 bits), `%si` (lower 16 bits), `%sil` (lower 8 bits)
- (and similar for `%rbp`, `%rsp`, `%rdi`)
- `%r9` (64-bit), `%r9d` (lower 32 bits), `%r9w` (lower 16 bits), `%r9b` (lower 8 bits)
- (and similar for `%r10` through `%r15`)

AT&T syntax reminder:

- `0x1234(%r9,%r10,4)` = memory at `0x1234` + `%r9` + `%r10` × 4
- `$0x12345678` = constant
- `0x12345678` = memory at `0x12345678`
- source, destination

virtual machines

illusion of dedicated machine

possibly different interface:

- system VM — interface looks like some physical machine
- system VM — OS runs inside VM
- process VM — what OS implements
- process VM — files instead of hard drives, threads instead of CPUs, etc.
- language VM — interface designed for particular programming language
- language VM — e.g. Java VM — knows about objects, methods, etc.

virtual machine implementation techniques

emulation:

- read instruction + giant if/else if/...

binary translation

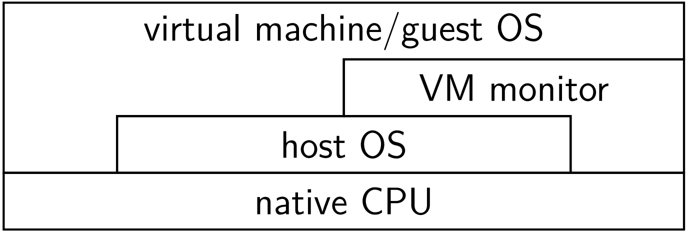
- compile machine code to new machine code

"native"

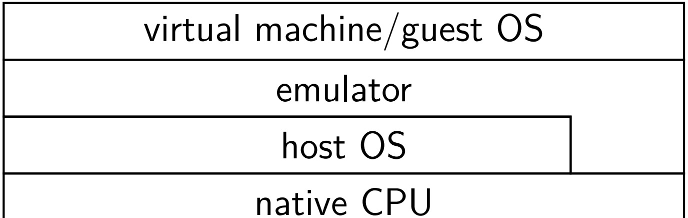
- run natively on hardware in user mode
- hardware triggers "exceptions" on special interrupts
- exceptions give VM implementation control

VM implementation strategies

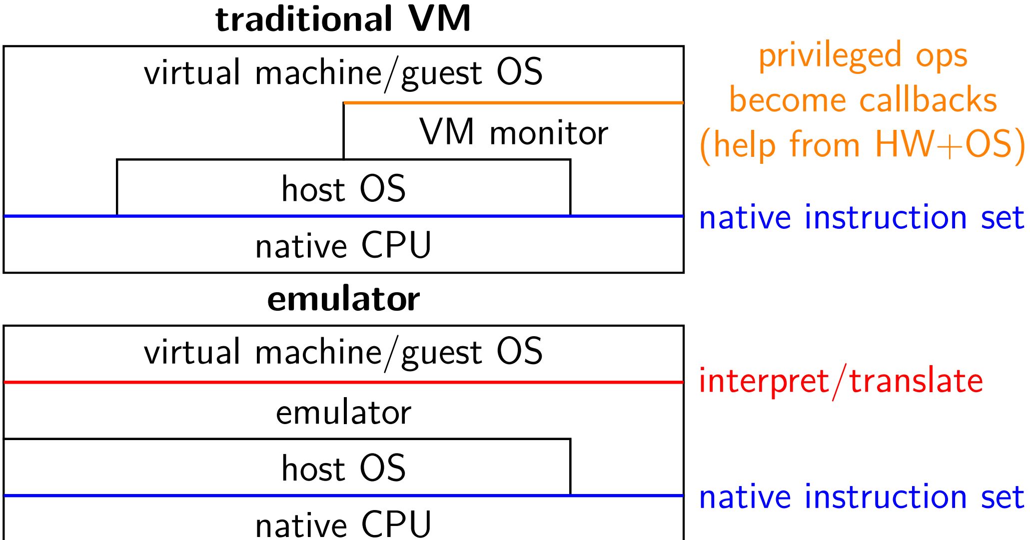
traditional VM



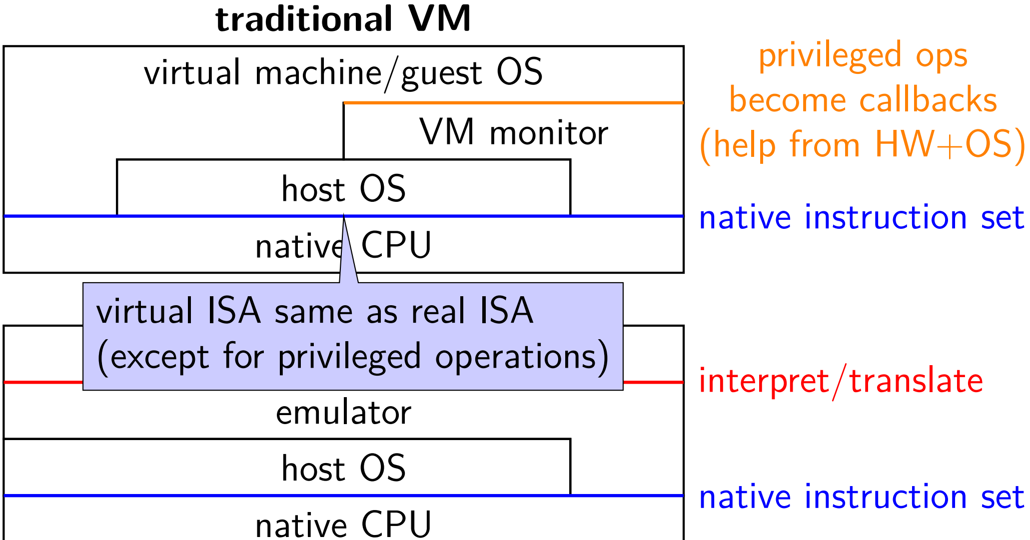
emulator



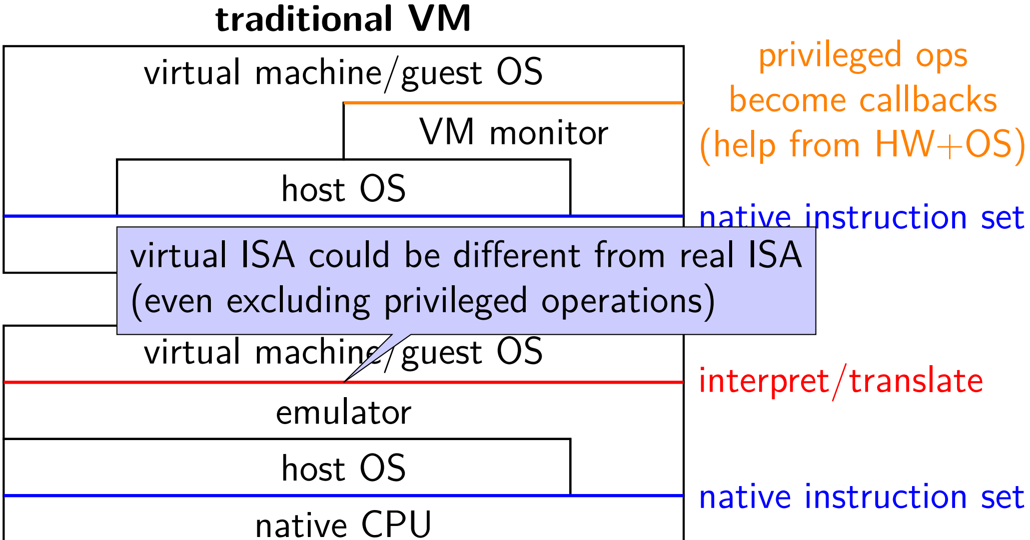
VM implementation strategies



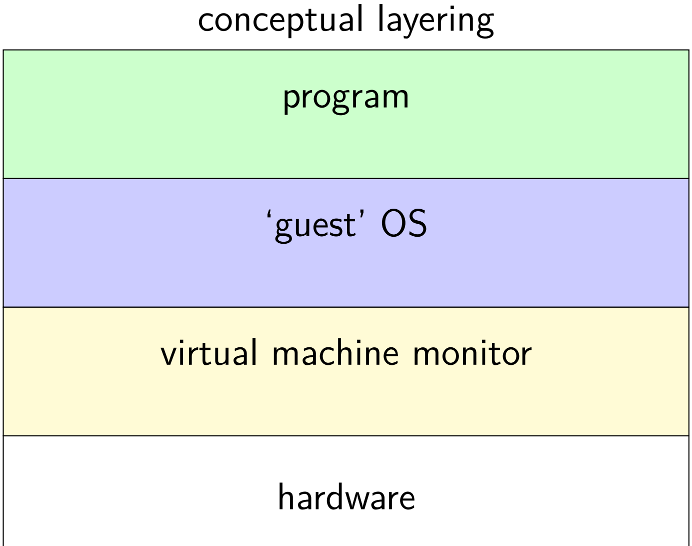
VM implementation strategies



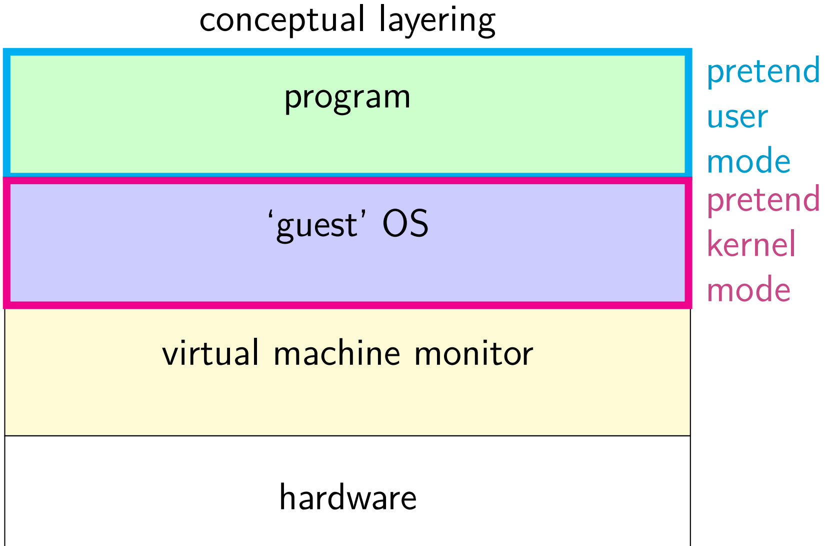
VM implementation strategies



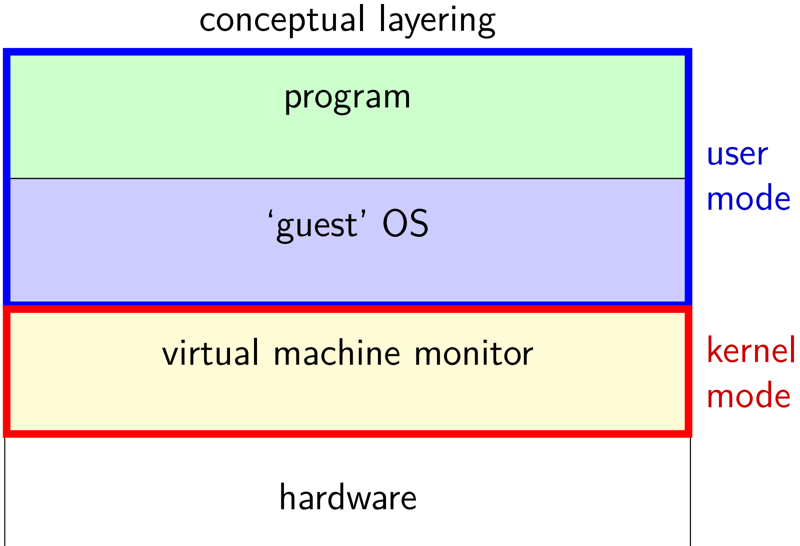
system call flow



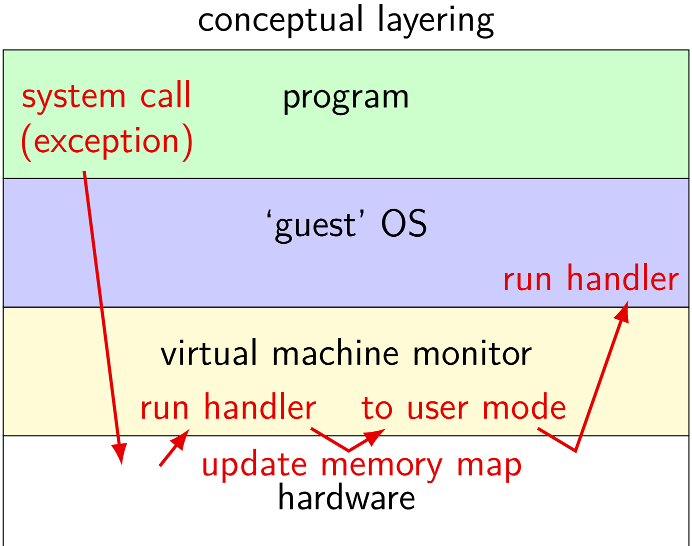
system call flow



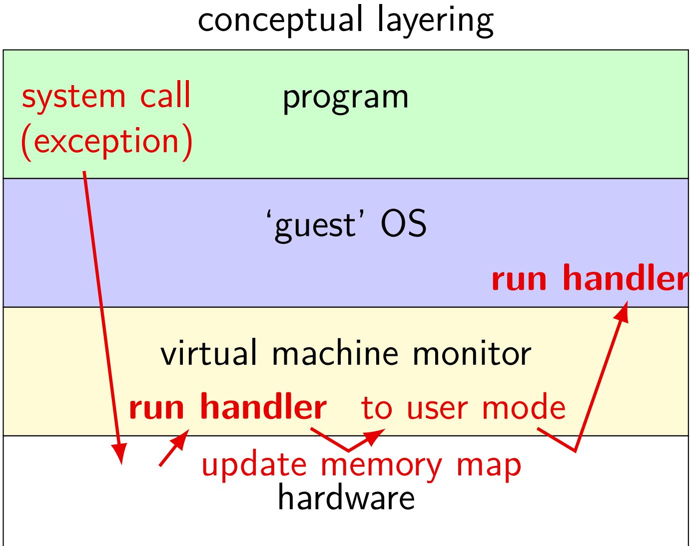
system call flow



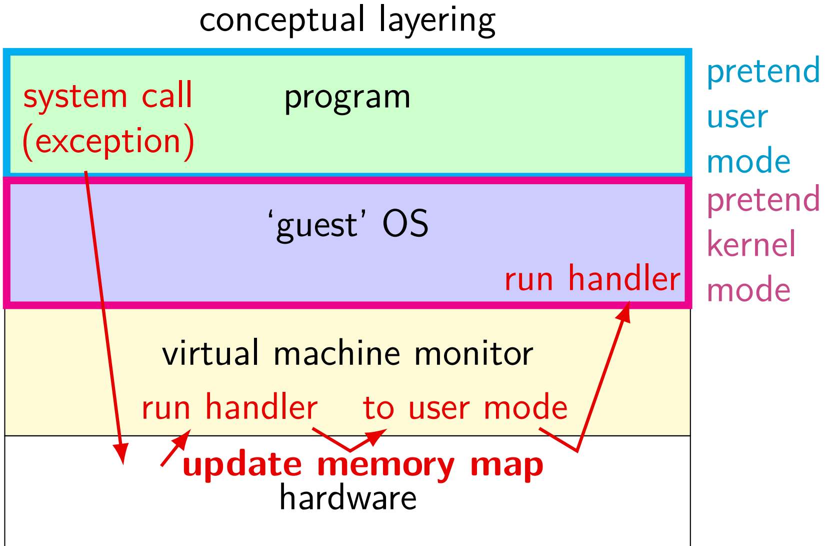
system call flow



system call flow



system call flow



VMs and malware

isolate malware from important stuff

sample malware behavior

- inspect memory for patterns — counter for metamorphic
- look for suspicious behavior generally

counter-VM techniques

detect VM-only devices

outrun patience of antivirus VM

unsupported instructions/system calls

…

debugger support

hardware support:

breakpoint instruction — debugger edits machine code to add

single-step flag — execute one instruction, jump to OS (debugger)

counter-debugger techniques

debuggers — also for analysis of malware

detect changes to machine code in memory

directly look for debugger

broken executables

...

AT&T syntax

`movq $42, 100(%rbx,%rcx,4)`

destination last

constants start with `$`; no `$` is an address

registers start with `%`

operand length (`q` = 8; `l` = 4; `w` = 2; `b` = 1)

`D(R1,R2,S)` = memory at `D + R1 + R2` × `S`

weird x86 features

segmentation: old way of dividing memory: `%fs:0x28`

- get segment # from FS register
- lookup that entry in a table
- add `0x28` to base adddress in table
- access memory as usual

rep prefix

- repeat instruction until rcx is 0
- …decrementing rcx each time

string instructions

- memory-to-memory; designed to be used with rep/etc. prefixes

executable/object file parts

type of file, entry point address, ...

seg#	file offset	memory loc.	size	permissions
1	`0x0123`	`0x3000`	`0x1200`	read/exec
2	`0x1423`	`0x5000`	`0x5000`	read/write

machine code + data for segments

symbol table: `foobar` at `0x2344`; `barbaz` at `0x4432`; ...
relocations: `printf` at `0x3333` (type: absolute); ...
section table, debug information, etc.

relocations?

unknown addresses — “holes” in machine code/etc.

linker lays out machine code

computes all symbol table addresses

uses symbol table addresses to fill in machine code

dynamic linking

executables not completely linked — library loaded at runtime

could use same mechanism, but ineffecient

instead: stubs:

```
0000000000400400 <puts@plt>:
  400400:       ff 25 12 0c 20 00           jmpq   *0x200c12(%rip)
                   /* 0x200c12+RIP = _GLOBAL_OFFSET_TABLE_+0x18 */
... later in main: ...
  40052d:       e8 ce fe ff ff              callq  400400 <puts@plt>
                       /* instead of call puts */
```

malware

evil software

various kinds:

- viruses
- worms
- trojan (horse)s
- potentially unwanted programs/adware
- rootkits
- logic bombs

worms

malicious program that copies itself

arranges to be run automatically (e.g. startup program)

may spread to other media (USB keys, etc.)

may spread over the network using vulnerabilities

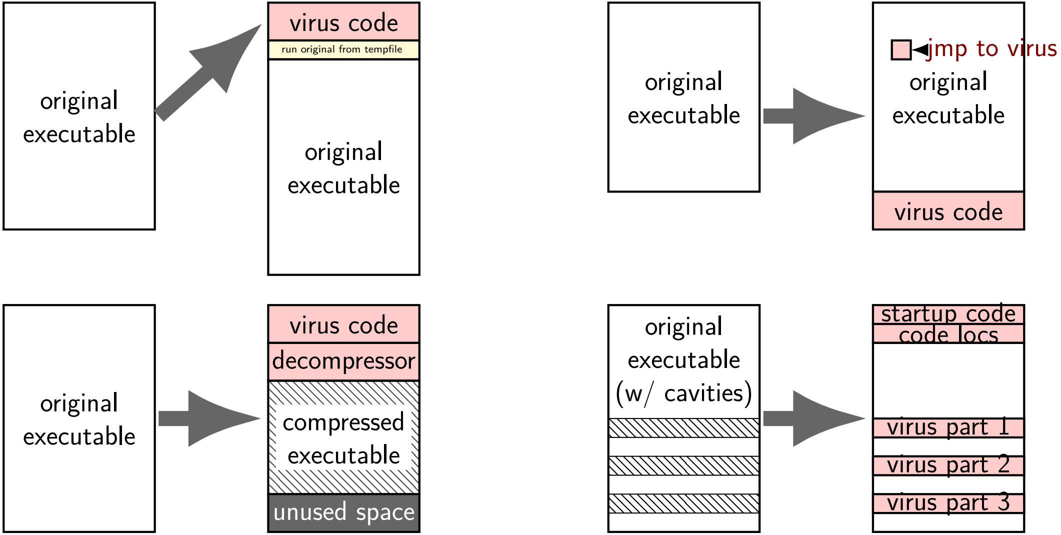
viruses

malware that embeds itself in innocent programs/files

spreads (primarily) by:

- hoping user shares infected files

code placement options



entry point choices

entry address

- perhaps a bit obvious

overwrite machine code and restore

edit call/jump/ret/etc.

- pattern-match for machine code
- in dynamic linking "stubs"
- in symbol tables
- call/ret at end of virus

pattern matching

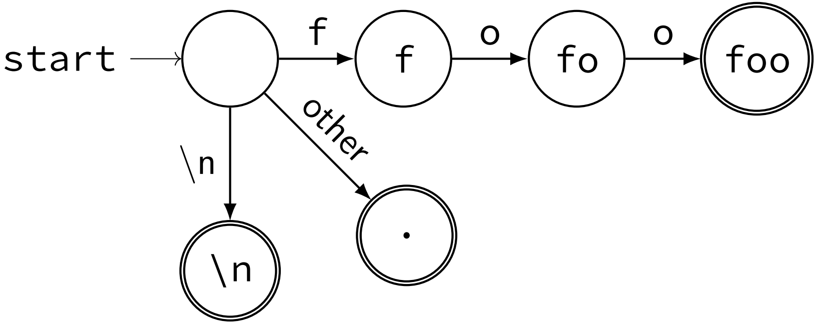
regular expressions — (almost) one-pass

fixed strings with "wildcards"

- addresses/etc. that change between instances of malware
- insert nops/variations on instructions

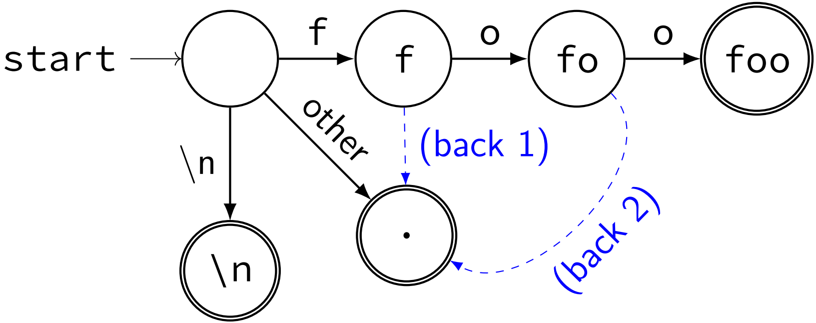
flex: state machines

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



flex: state machines

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



behavior-based detection/blocking

modifying executables? etc.

must be malicious

armored viruses, etc.

evade analysis:

- “encrypt” code (break disassembly)
- detect/break debuggers
- detect/break VMs

evade signatures:

- oligomorphic/polymorphic: varying “decrypter”
- metamorphic: varying “decrypter” and varying “encrypted” code

evade active detection:

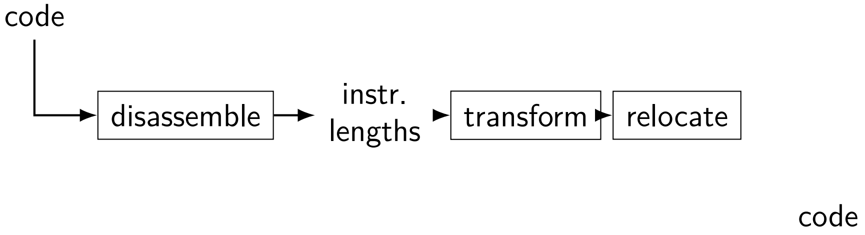
- tunnelling — skip anti-virus hooks
- stealth — ‘hook’ system calls to say “executable/etc. unchanged”
- retroviruses — break/uninstall/etc. anti-virus software

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



hooking mechanisms

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

- e.g. creating new files

ideal mechanism: OS support

less ideal mechanism: change library loading

- e.g. replace 'open', 'fopen', etc. in libraries

less ideal mechanism: replace OS exception (system call) handlers

- very OS version dependent

software vulnerabilities

unintended program behavior an adversary can use

memory safety bugs

- especially buffer overflows

not checking inputs/permissions

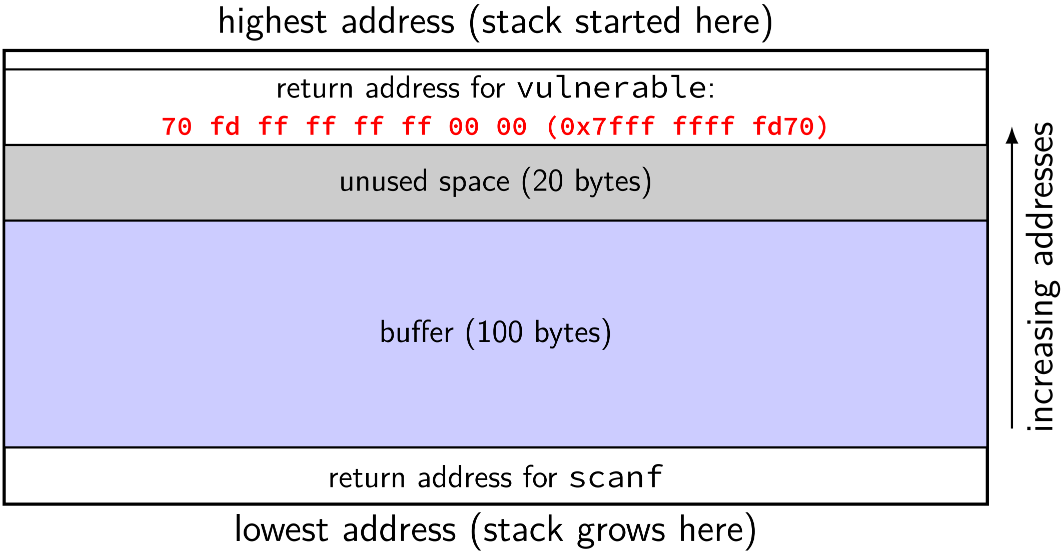
injection/etc. bugs

exploits

something that uses a vulnerability to do something

example: stack smashing — exploit for stack buffer overflows

return-to-stack



return-to-stack