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! */
}
```

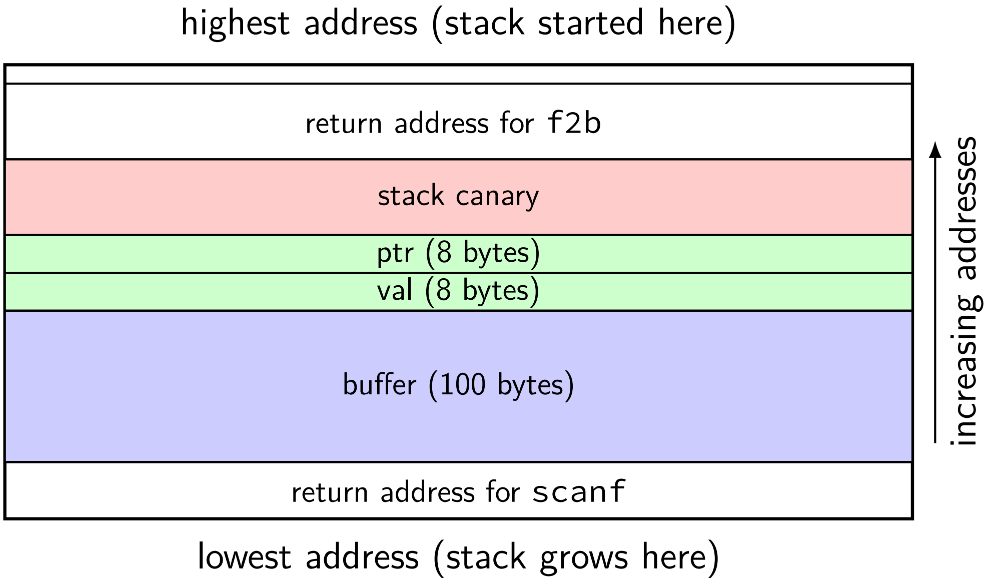
adapted from Pincus and Baker, Figure 2

pointer subterfuge

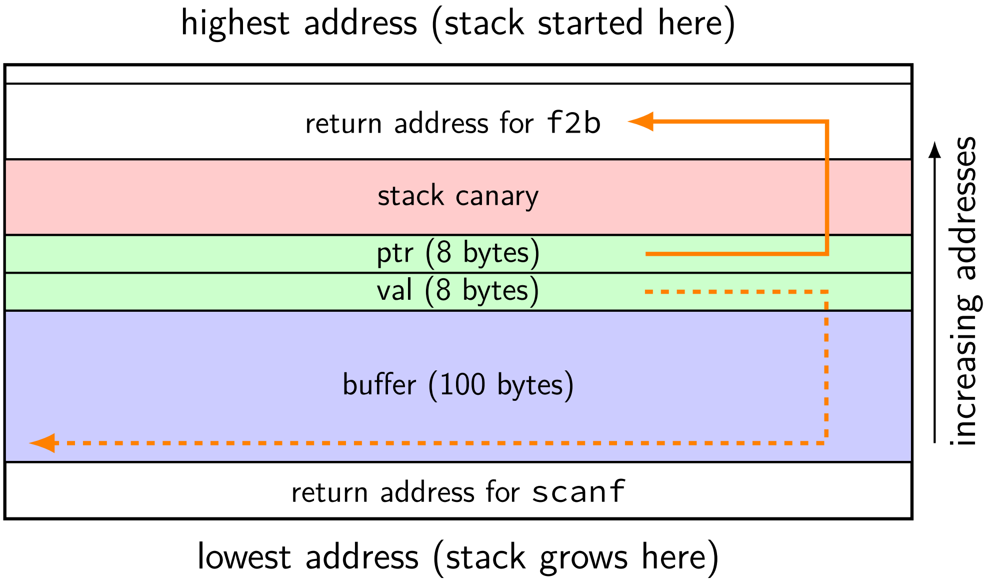
```
void f2b(void *arg, size_t len) {
    char buffer[100];
    long val = ...; /* assume on stack */
    long *ptr = ...; /* assume on stack */
    memcpy(buff, arg, len); /* overwrite ptr? */
    *ptr = val; /* arbitrary memory write! */
}
```

adapted from Pincus and Baker, Figure 2

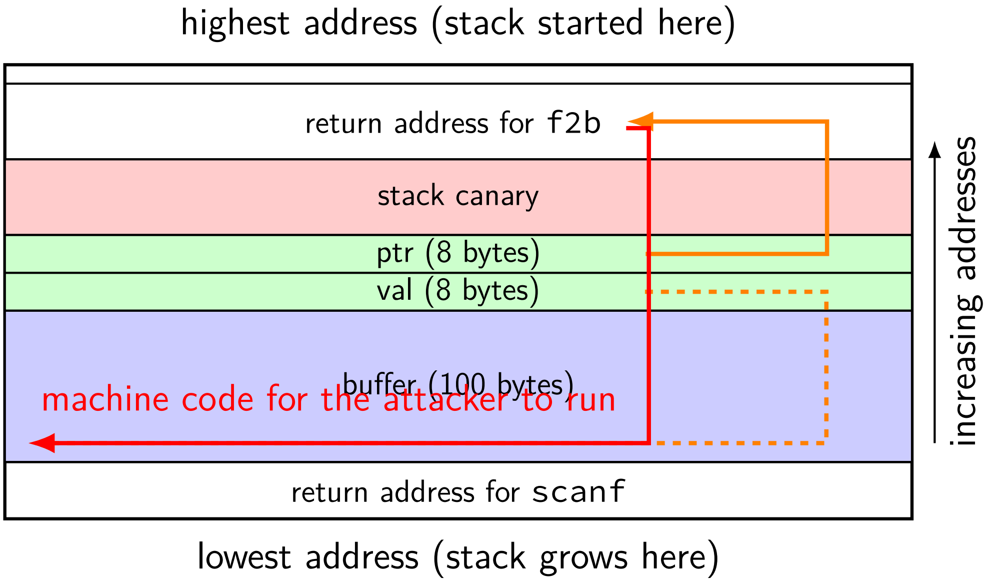
skipping the canary



skipping the canary



skipping the canary



beyond return addresses

pointer subterfuge let us overwrite anything

my example: showed return address

but return address is tricky to locate exactly

but there are *easier options!*

arbitrary memory write

bunch of scenarios that lead to *single arbitrary memory write*

- format exploits are one, but we'll find more!!

typical result: arbitrary code execution

how?

arbitrary memory write

bunch of scenarios that lead to *single arbitrary memory write*

- format exploits are one, but we'll find more!!

typical result: arbitrary code execution

how?

overwrite existing machine code (insert jump?)

- problem: usually not writable

overwrite return address directly

- observation: don't care about stack canaries — skip them

overwrite other function pointer?

overwrite another data pointer — copy more?

arbitrary memory write

bunch of scenarios that lead to *single arbitrary memory write*

- format exploits are one, but we'll find more!!

typical result: arbitrary code execution

how?

overwrite existing machine code (insert jump?)

- problem: usually not writable

overwrite return address directly

- observation: don't care about stack canaries — skip them

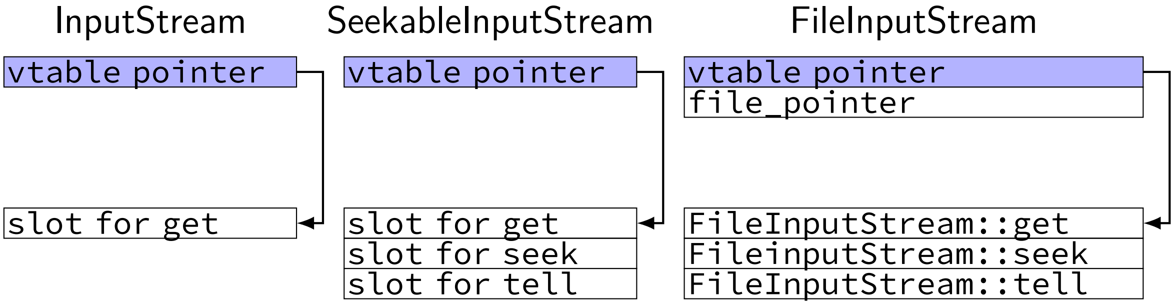
overwrite other function pointer?

overwrite another data pointer — copy more?

C++ inheritence

```
class InputStream {
public:
    virtual int get() = 0;
    // Java: abstract int get();
    ...
};
class SeekableInputStream : public InputStream {
public:
    virtual void seek(int offset) = 0;
    virtual int tell() = 0;
};
class FileInputStream : public InputStream {
public:
    int get();
    void seek(int offset);
    int tell();
    ...
};
```

C++ inheritence: memory layout



C++ implementation (pseudo-code)

```
struct InputStream_vtable {
    int (*get)(InputStream* this);
};

struct InputStream {
    InputStream_vtable *vtable;
};

...

    InputStream *s = ...;
    int c = (s->vtable->get)(s);
```

C++ implementation (pseudo-code)

```
struct SeekableInputStream_vtable {
    struct InputStream_vtable as_InputStream;
    void (*seek)(SeekableInputStream* this, int offset);
    int (*tell)(SeekableInputStream* this);
};

struct FileInputStream {
    SeekableInputStream_vtable *vtable;
    FILE *file_pointer;
};

...

    FileInputStream file_in = { the_FileInputStream_vtable,  ... };
    InputStream *s = (InputStream*) &file_in;
```

C++ implementation (pseudo-code)

```
SeekableInputStream_vtable the_FileInputStream_vtable = {
    &FileInputStream_get,
    &FileInputStream_seek,
    &FileInputStream_tell,
};

...

    FileInputStream file_in = { the_FileInputStream_vtable,  ... };
    InputStream *s = (InputStream*) &file_in;
```

attacking function pointer tables

option 1: overwrite table entry directly

- required/easy for Global Offset Table — fixed location
- usually not possible for VTables — read-only memory

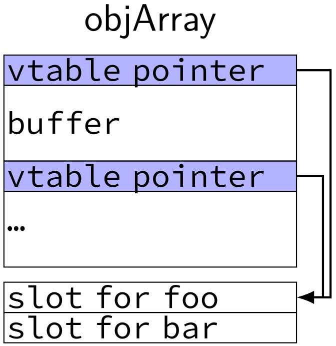
option 2: create table in buffer (big list of pointers to shellcode), point to buffer

- useful when table pointer next to buffer
- (e.g. C++ object on stack next to buffer)

option 3: find suitable pointer elsewhere

- e.g. point to wrong part of vtable to run different function

exercise



```
class VulnerableClass {
public:
    char buffer[100];
    virtual void foo();
    virtual void bar();
};
VulnerableClass objArray[10];
```

if we can overflow `objArray[0].buffer` to change array[1]'s vtable pointer and know array[1].foo() will be called; finish the plan:

buffer[0]: ____________

buffer[50]: ____________

array[1]'s vtable pointer: ____________

A. shellcode

B. address of buffer[0]

C. address of buffer[50]

D. address of original vtable

arbitrary memory write

bunch of scenarios that lead to *single arbitrary memory write*

- format exploits are one, but we'll find more!!

typical result: arbitrary code execution

how?

overwrite existing machine code (insert jump?)

- problem: usually not writable

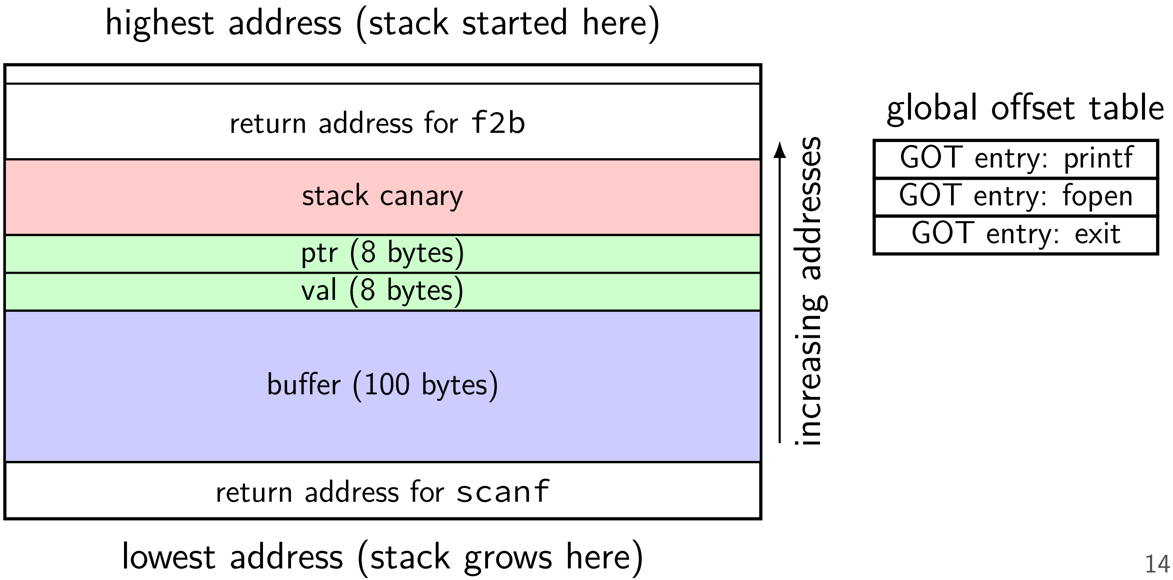
overwrite return address directly

- observation: don't care about stack canaries — skip them

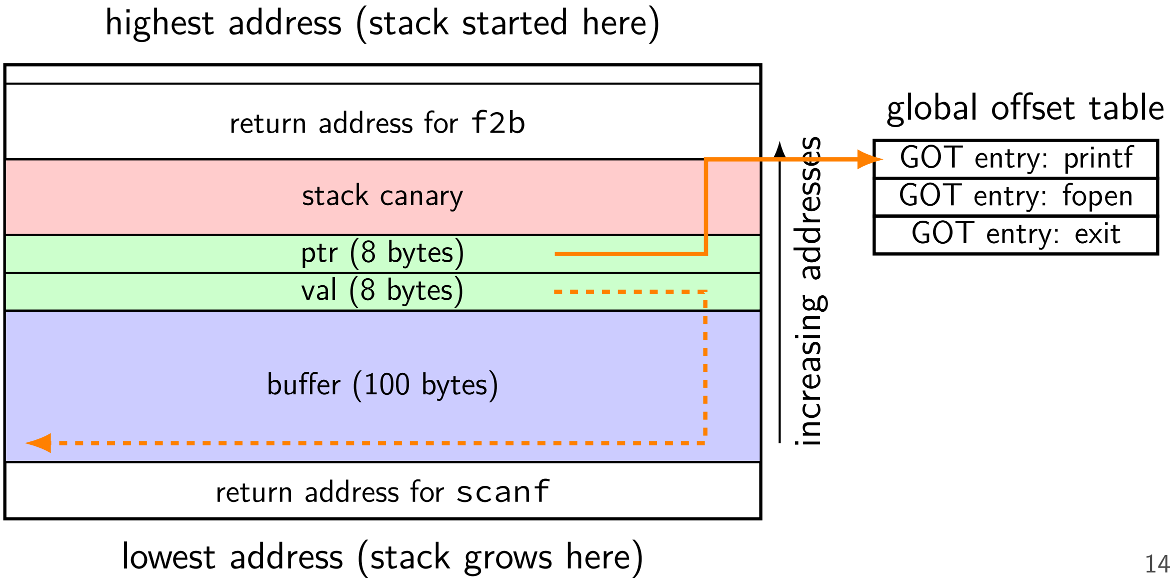
overwrite other function pointer?

overwrite another data pointer — copy more?

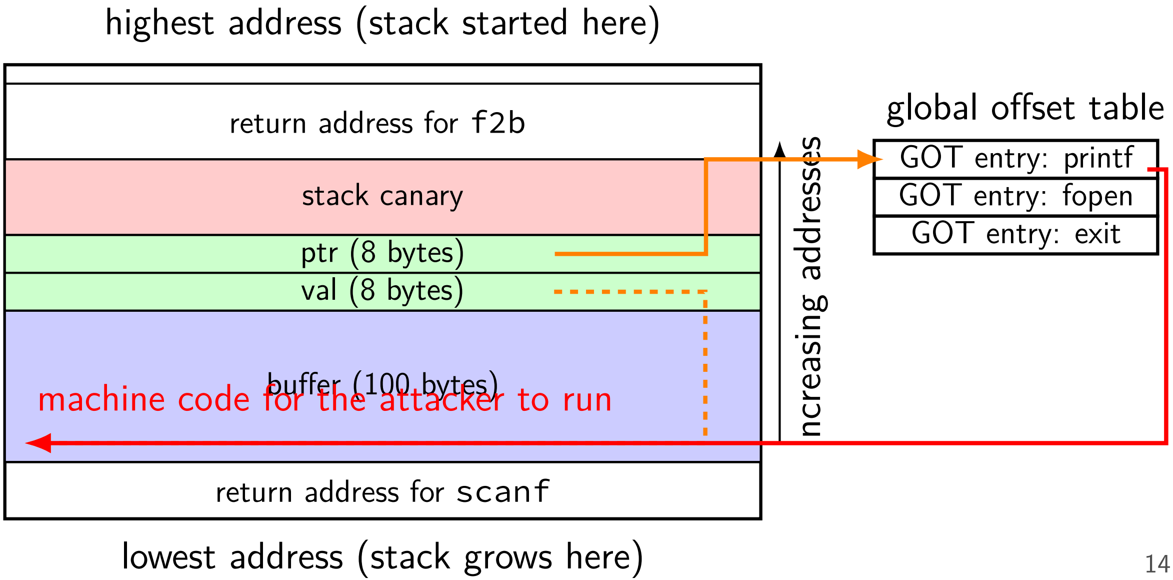
attacking the GOT



attacking the GOT

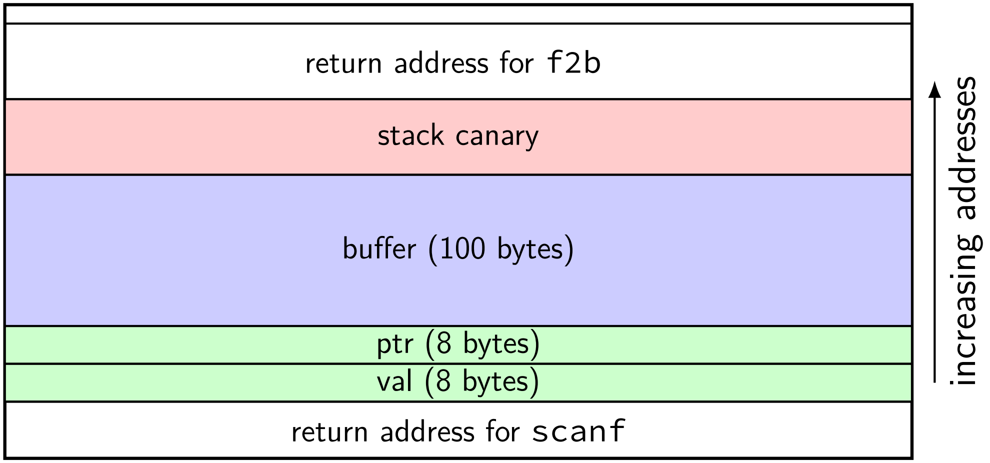


attacking the GOT



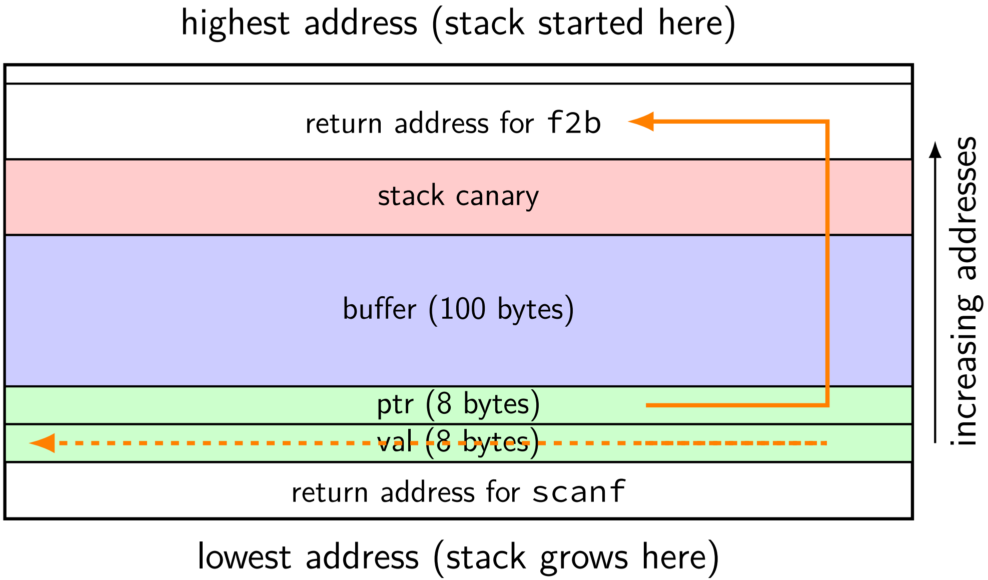
laying out stack to avoid subterfuge

highest address (stack started here)



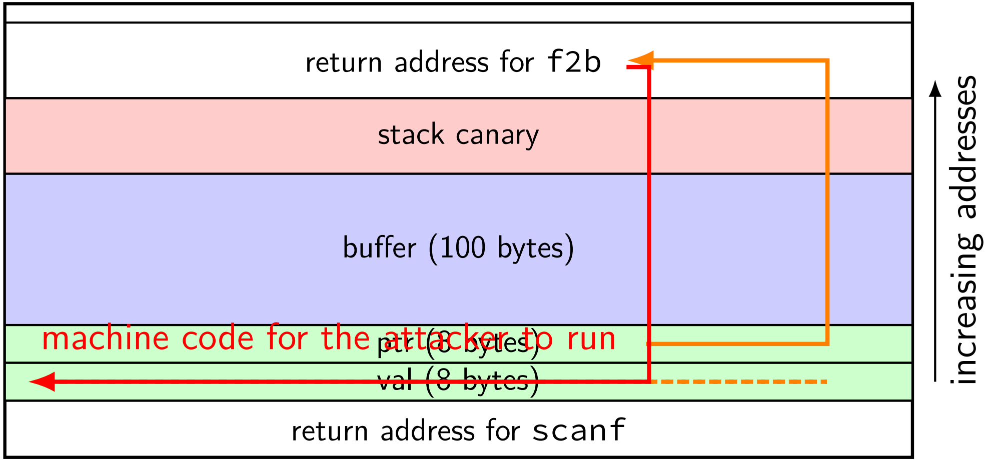
lowest address (stack grows here)

laying out stack to avoid subterfuge



laying out stack to avoid subterfuge

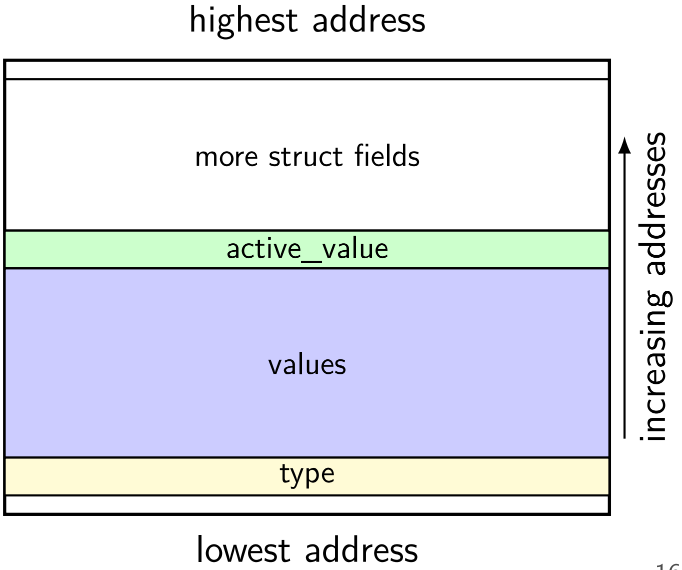
highest address (stack started here)



lowest address (stack grows here)

other subterfuge cases (1)

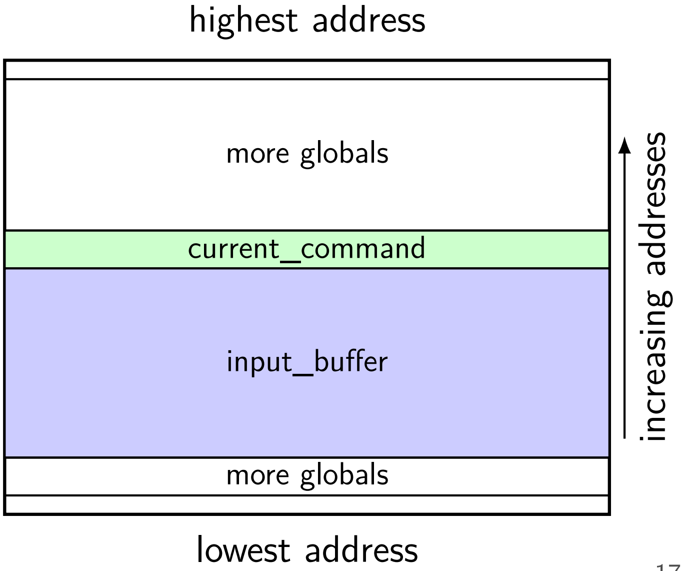
```
struct Command {
  CommandType type;
  int values[MAX_VALUES];
  int *active_value;
  ...
};
```



other subterfuge cases (2)

```
Command *current_command;
char input_buffer[4096];

void run_next_command() {
  if (!current_command) {
    current_command =
        getNext();
  }
  current_command-> ...
  ...
}
```

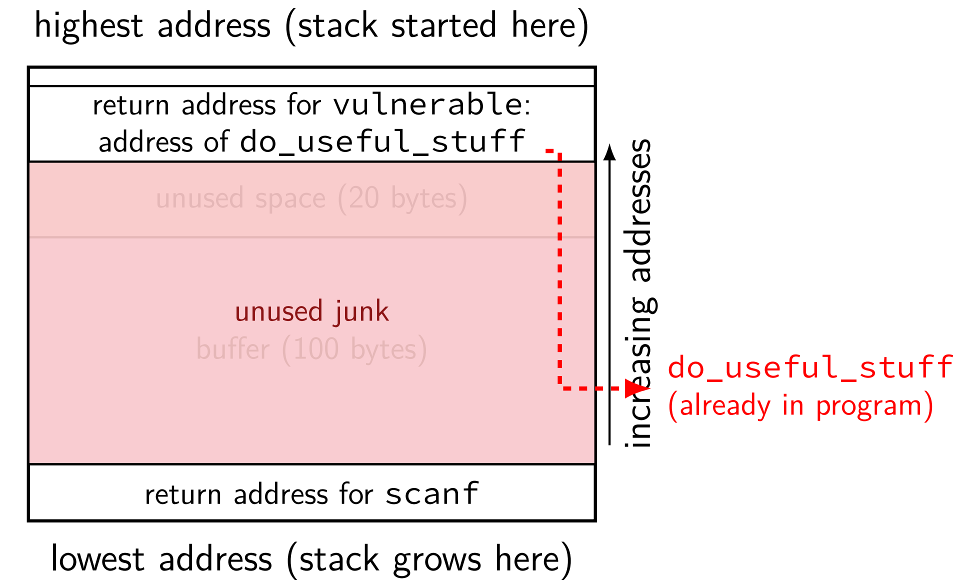


so far overwrites

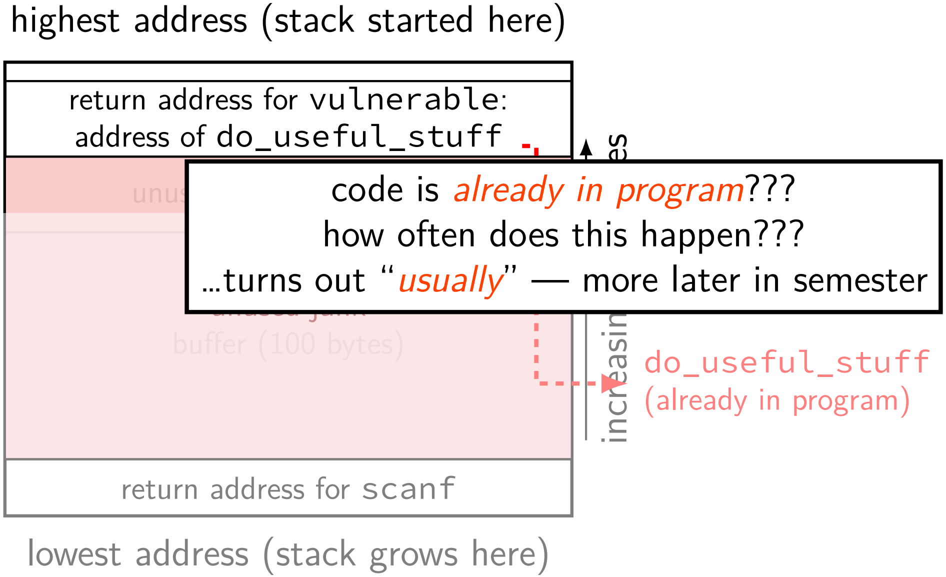
once we found a way to overwrite function pointer
easiest solution seems to be: direct to our code

...but alterante places to direct it to

return-to-somewhere



return-to-somewhere



highest address (stack started here)
return address for vulnerable:
address of do_useful_stuff
unus
unused junk
buffer (100 bytes)
increasing addresses
do_useful_stuff
(already in program)
return address for scanf
lowest address (stack grows here)

code is *already in program*???

how often does this happen???

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

example: system()

```
NAME
       system - execute a shell command

SYNOPSIS
       #include <stdlib.h>

       int system(const char *command);
```

part of C standard library

in any program that dynamically links to libc

challenge: need to hope argument register (rdi) set usefully

locating system() Linux

```
$ ldd /bin/ls
        linux-vdso.so.1 (0x00002aaaaaade000)
        libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x00002aaaaab3a000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00002aaaaab65000)
        libpcre2-8.so.0 => /usr/lib/x86_64-linux-gnu/libpcre2-8.so.0 (0x00002aaaaad57000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00002aaaaade7000)
        /lib64/ld-linux-x86-64.so.2 (0x00002aaaaaaab000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00002aaaaaded000)
$ objdump --dynamic-syms /lib/x86_64-linux-gnu/libc.so.6 | grep system
0000000000156a80 g    DF .text  0000000000000067  GLIBC_2.2.5 svcerr_systemerr
0000000000055410 g    DF .text  000000000000002d  GLIBC_PRIVATE __libc_system
0000000000055410  w   DF .text  000000000000002d  GLIBC_2.2.5 system
```

if address randomization disabled:
address should be 0x00002aaaaab650 + 0x55410

`ldd` — "what libraries does this load and where?"

similar tools for other OSes

case study (simplified)

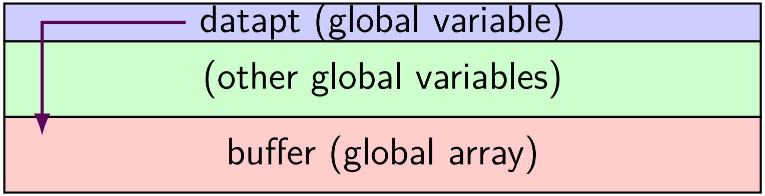
bug in NTPd (Network Time Protocol Daemon)

via Stephen Röttger, “Finding and exploiting ntpd vulnerabilities”
    https://googleprojectzero.blogspot.com/2015/01/finding-and-exploiting-ntpd.html

```
static void
ctl_putdata(
  const char *dp,
  unsigned int dlen,
  int bin   /* set to 1 when data is binary */
  ) {
    ...
    memmove((char *)datapt, dp, (unsigned)dlen);
    datapt += dlen;
    datalinelen += dlen;
```

the target

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

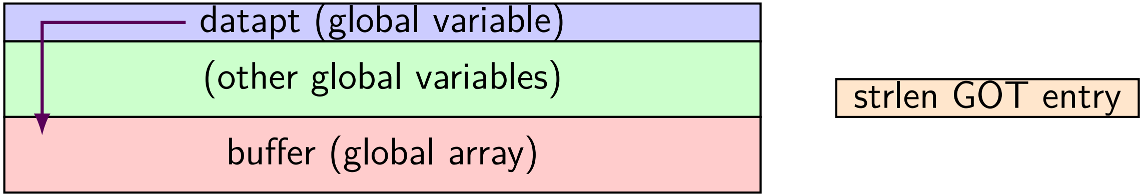


more context

```
memmove((char *)datapt, dp, (unsigned)dlen);
...
...
strlen(some_user_supplied_string)
/* calls strlen@plt
   looks up global offset table entry! */
```

the target

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



overall exploit

overwrite `datapt` to point to strlen GOT entry

overwrite value of strlen GOT entry

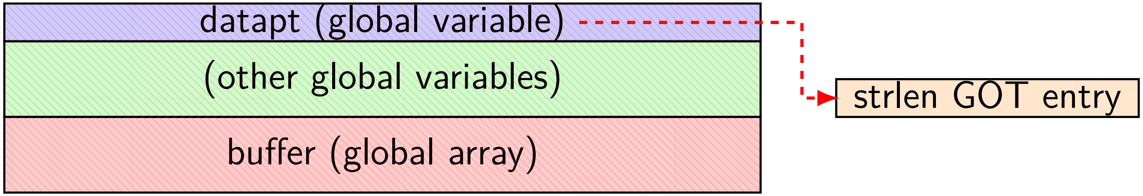
example target: `system` function

  executes command-line command specified by argument

supply string to provide argument to "`strlen`"

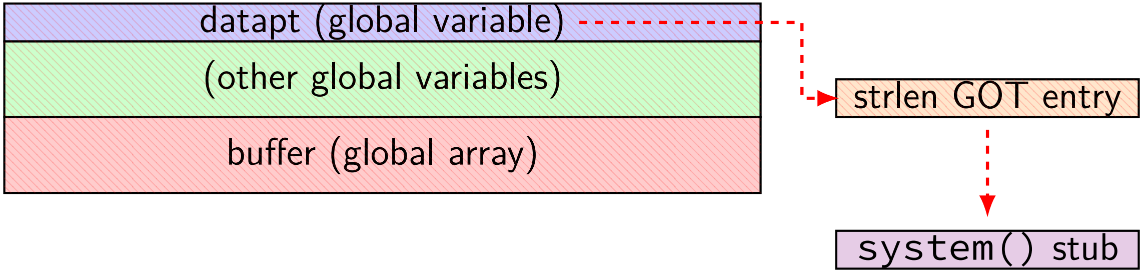
the target

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



the target

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



overall exploit: reality

real exploit was more complicated

needed to defeat more mitigations

needed to deal with not being able to write `\0`

actually tricky to send things that trigger buffer write
    (meant to be local-only)

subterfuge exercise

```
struct Student {
    char email[128];
    struct Assignment *assignments[16];
    ...
};
struct Assignment {
    char submission_file[128];
    char regrade_request[1024];
    ...
};
void SetEmail(Student *s, char *new_email) { strcpy(s->email, new_email); }
void AddRegradeRequest(Student *s, int index, char *request) {
    strcpy(s->assignments[index]->regrade_request, request);
}
void vulnerable(char *STRING1, char *STRING2) {
    SetEmail(s, STRING1); AddRegradeRequest(s, 0, STRING2);
}
```

exercise: to set `0x1020304050` to `0xAABBCCDD`, what should STRING1, STRING2 be?

(assume 64-bit pointers, no padding in structs, little-endian)

backup slides