Case study: VSFTPD

Very Secure FTPD

- **FTP**: File Transfer Protocol
  - More popular before the rise of HTTP, but still in use
  - 90's and 00's: **FTP daemon compromises were frequent and costly**, e.g., in Wu-FTPD, ProFTPd, …
- **Very thoughtful design** aimed to **prevent** and **mitigate security defects**
- But also to **achieve good performance**
  - Written in C
- Written and maintained by Chris Evans since 2002
  - **No security breaches that I know of**

https://security.appspot.com/vsftpd.html

VSFTPD Threat model

- **Clients untrusted, until authenticated**

- Once authenticated, **limited** trust:
  - According to user's **file access control policy**
  - For the files being served FTP (and not others)

- Possible attack goals
  - **Steal** or **corrupt resources** (e.g., files, malware)
  - **Remote code injection**

- Circumstances:
  - **Client attacks server**
  - **Client attacks** another **client**

Defense: Secure Strings

```
struct mystr
{
  char* PRIVATE HANDS OFF p buf;
  unsigned int PRIVATE_HANDS_OFF_len;
  unsigned int PRIVATE_HANDS_OFF_alloc_bytes;
};
```

Normal (zero-terminated) C string

The actual length (i.e., `strlen(PRIVATE_HANDS_OFF_p_buf)`)

Size of buffer returned by `malloc`

```
void
private_str_alloc_memchunk(struct mystr* p_str, const char* p_src,
                           unsigned int len)
{
  ...
}


void
str_copy(struct mystr* p_dest, const struct mystr* p_src)
{
  private_str_alloc_memchunk(p_dest, p_src->p_buf, p_src->len);
}
```

```
struct mystr
{
  char* p_buf;
  unsigned int len;
  unsigned int alloc_bytes;
};
```

replace uses of `char*` with `struct mystr*`
and uses of `strcpy` with `str_copy`

```
void
private_str_alloc_memchunk(struct mystr* p_str, const char* p_src,
                           unsigned int len)
{
  /* Make sure this will fit in the buffer */
  unsigned int buf_needed;
  if (len + 1 < len)
  {
    bug("integer overflow");
  }
  buf_needed = len + 1;
  if (buf_needed > p_str->alloc_bytes)
  {
    str_free(p_str);
    s_setbuf(p_str, vsf_sysutil_malloc(buf_needed));
    p_str->alloc_bytes = buf_needed;
  }
  vsf_sysutil_memcpy(p_str->p_buf, p_src, len);
  p_str->p_buf[len] = '\0';
  p_str->len = len;
}
```

consider NUL terminator when computing space

allocate space, if needed

copy in p_src contents

```
struct mystr
{
  char* p_buf;
  unsigned int len;
  unsigned int alloc_bytes;
};
```

Copy in at most `len` bytes from `p_src` into `p_str`

Defense: Secure Stdcalls

- Common problem: **error handling**
  - Libraries **assume** that **arguments are well-formed**
  - Clients **assume** that library **calls always succeed**

- Example: `malloc()`
  - What if argument is non-positive?
    - We saw earlier that integer overflows can induce this behavior
    - Leads to buffer overruns
  - What if returned value is NULL?
    - Oftentimes, a deference means a crash
    - On platforms without memory protection, a dereference can cause corruption

```c
void*
vsf_sysutil_malloc(unsigned int size)
{
  void* p_ret;
  /* Paranoia - what if we got an integer overflow/underflow? */
  if (size == 0 || size > INT_MAX)
  {
    bug("zero or big size in vsf_sysutil_malloc");
  }
  p_ret = malloc(size);
  if (p_ret == NULL)
  {
    die("malloc");
  }
  return p_ret;
}
```

fails if it receives malformed argument or runs out of memory

Defense: Minimal Privilege

- **Untrusted input** always handled by **non-`root` process**
  - Uses IPC to delegate high-privilege actions
    - Very little code runs as `root`

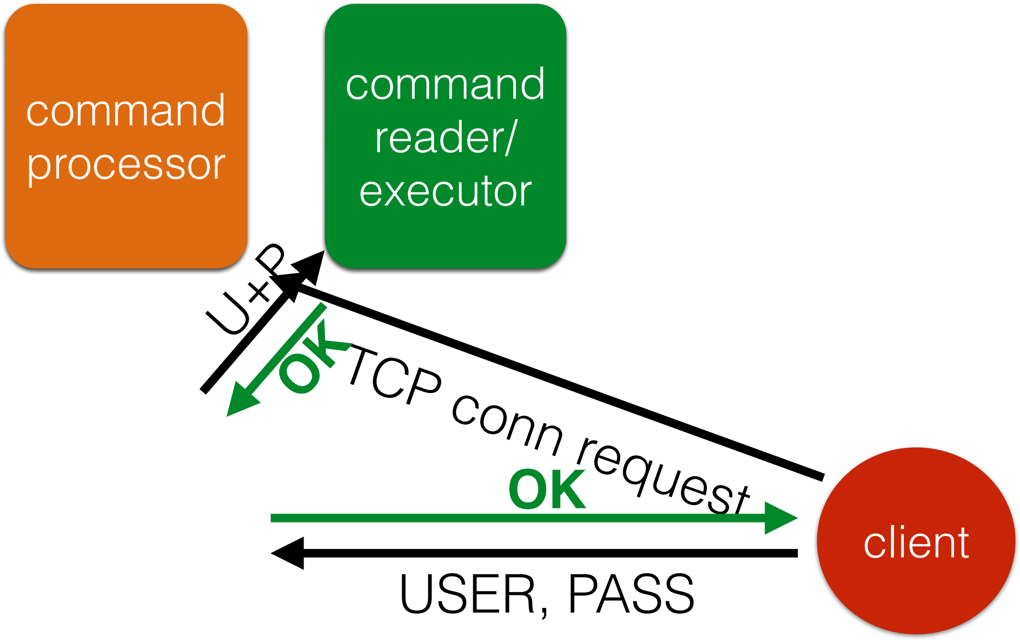
small trusted computing base

- **Reduce privileges** as much as possible
  - Run as particular (unprivileged) user
    - File system access control enforced by OS
  - Use capabilities and/or SecComp on Linux
    - Reduces the system calls a process can make

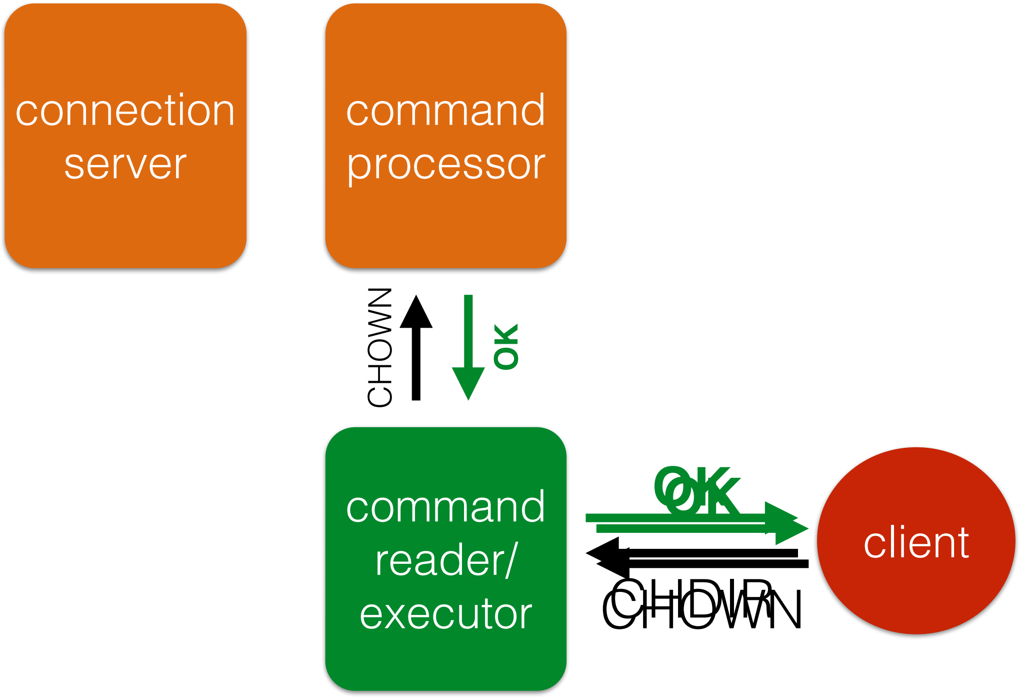
- **`chroot` to hide all directories** but the current one
  - Keeps visible only those files served by FTP

principle of least privilege

Connection Establishment



Performing Commands



connection server

command processor

CHOWN

OK

command reader/ executor

OK

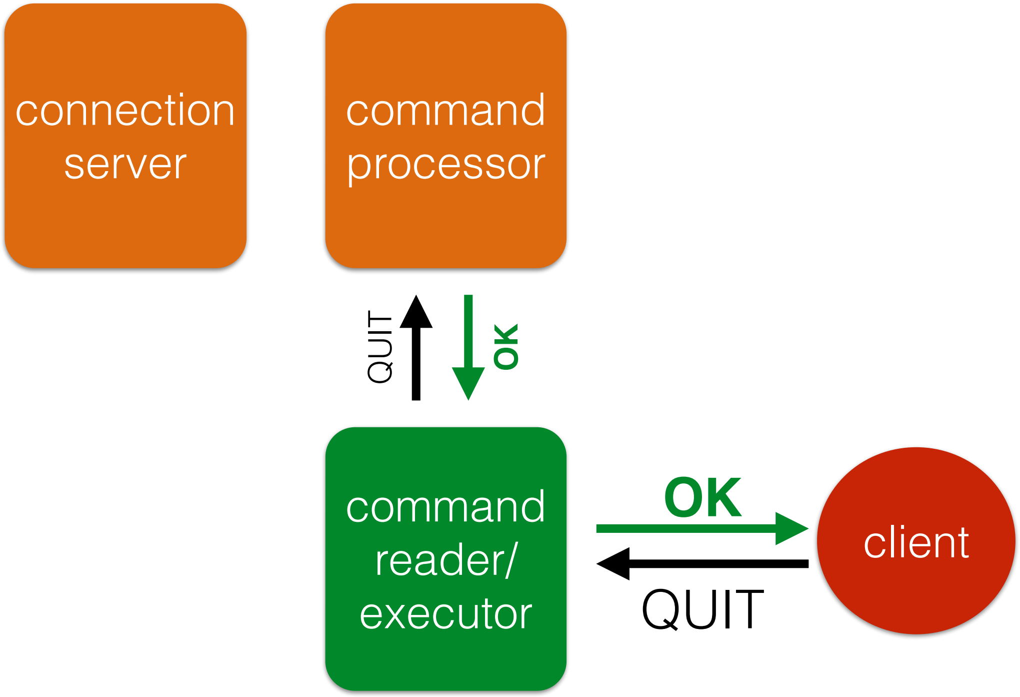
OK

CHDIR

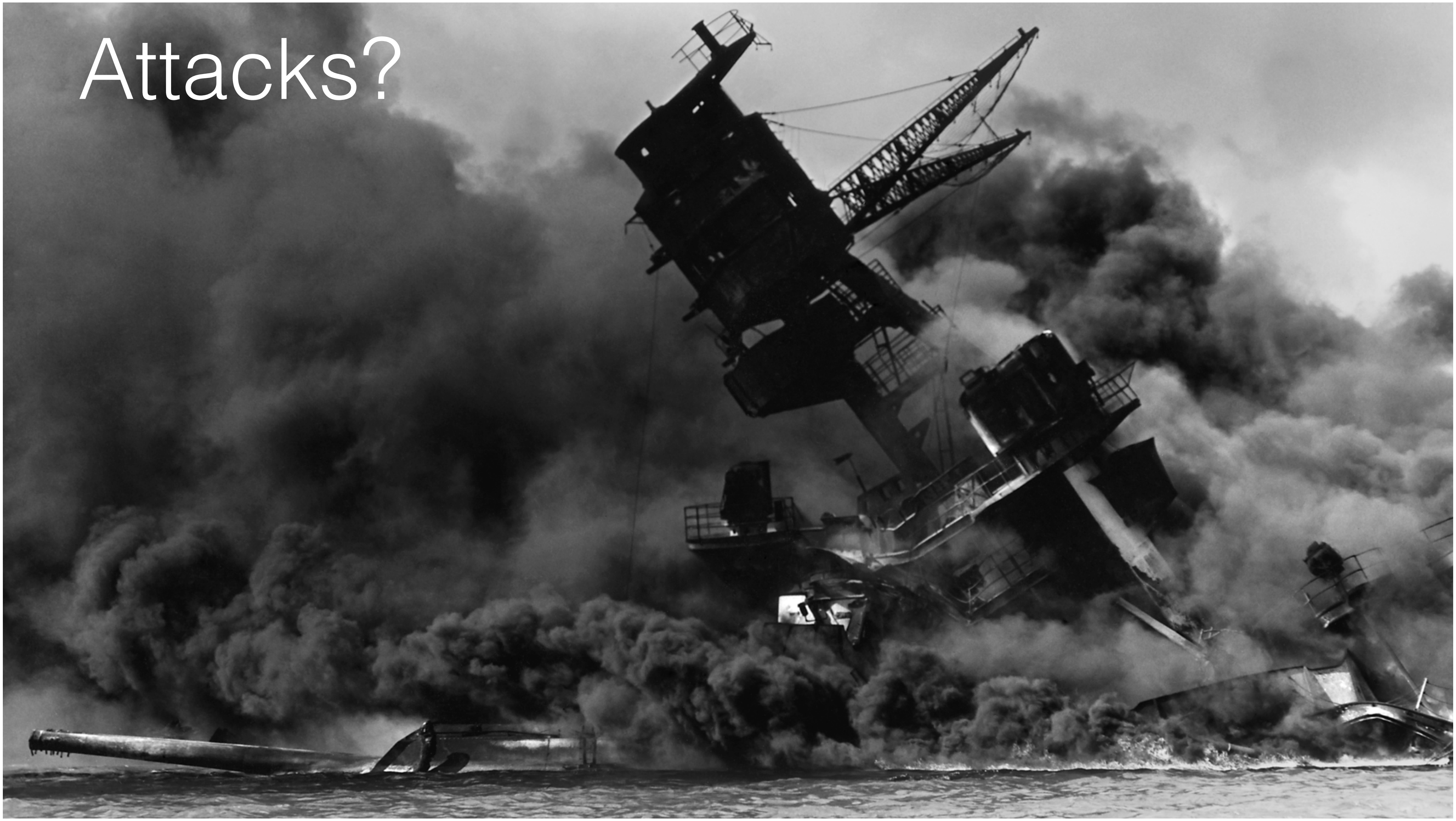
CHOWN

client

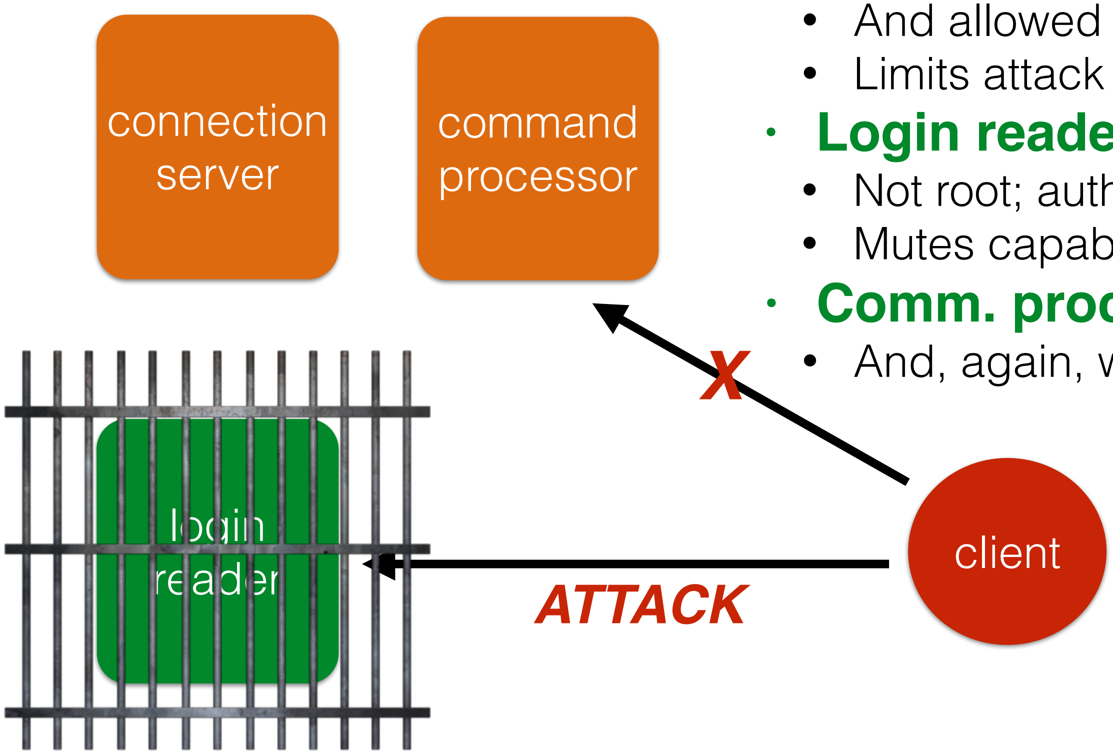
Logging out


connection server
command processor
QUIT
OK
command reader/ executor
OK
QUIT
client

Attacks?

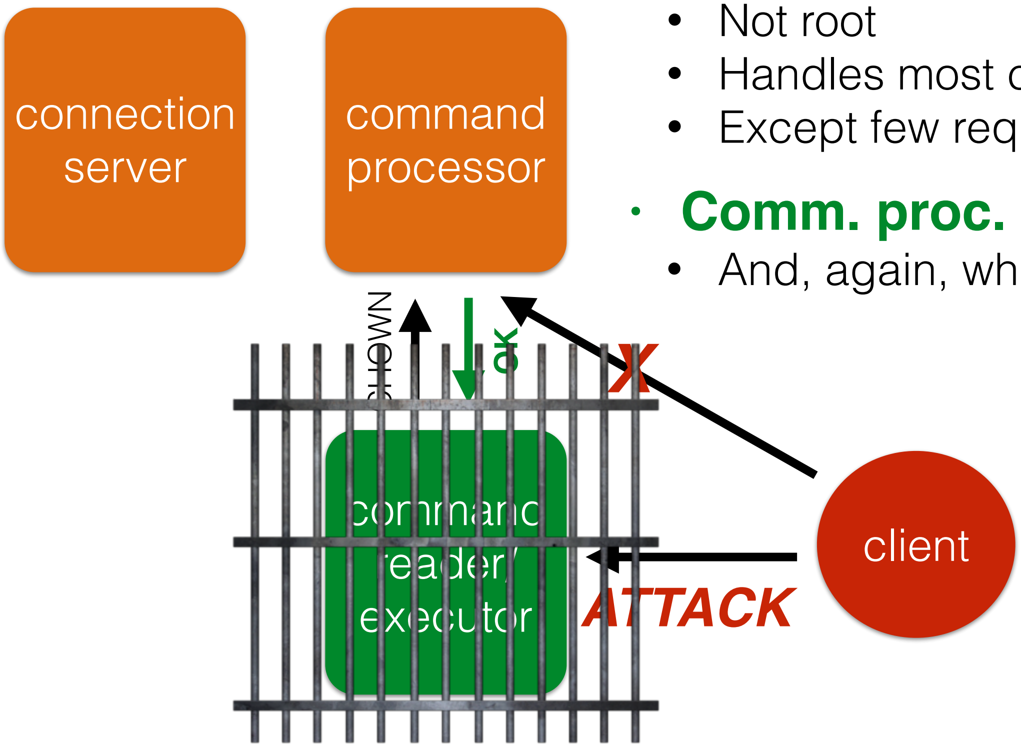


Attack: Login



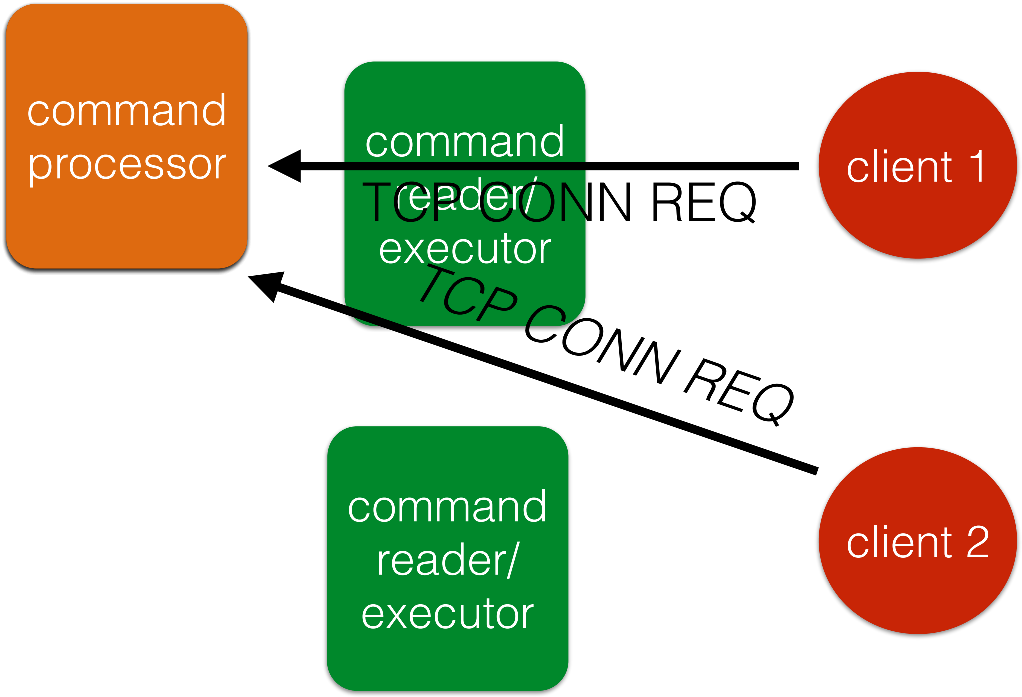
- **Login reader white-lists input**
  - And allowed input very limited
  - Limits attack surface
- **Login reader has limited privilege**
  - Not root; authentication in separate process
  - Mutes capabilities of injected code
- **Comm. proc. only talks to reader**
  - And, again, white-lists its limited input

Attack: Commands

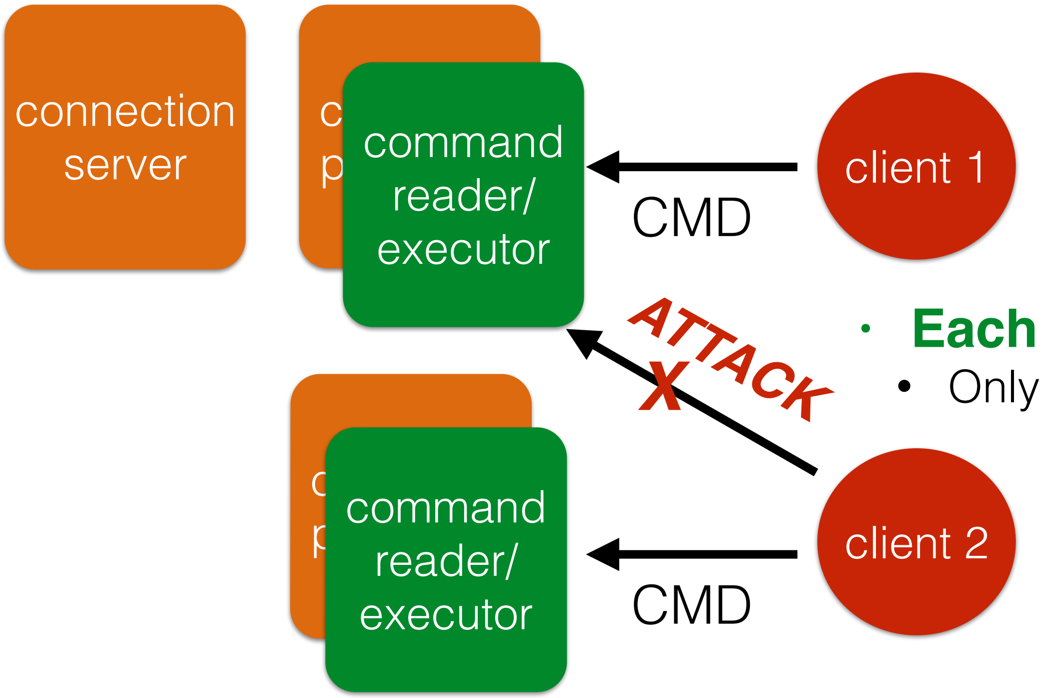


- **Command reader sandboxed**
  - Not root
  - Handles most commands
  - Except few requiring privilege
- **Comm. proc. only talks to reader**
  - And, again, white-lists its limited input

Attack: Cross-session



Attack: Cross-session



- **Each session isolated**
  - Only can talk to one client

Other VSFTPD notables

- **Secure sockets** option, for encrypted connections
  - But **not turned on by default**: “OpenSSL is a massive quantity of code which is essentially parsing complex protocol under the full control of remote malicious clients. SSL / TLS is disabled by default, both at compile time and run time. This forces packagers and administrators to make the decision that they trust the OpenSSL library. I personally haven't yet formed an opinion on whether I consider the OpenSSL code trustworthy.”
- **Eschews trusting other executables**
  - Doesn’t use `/bin/ls` for directory listings

The rest of the process

- Four common development phases:
  - **Requirements**
  - **Design**
  - **Implementation**
  - **Testing/assurance**

- **Up next:**
  - Automated code review using **static analysis**
  - “Whitebox fuzz testing” using **symbolic execution**
  - **Penetration testing** with tools and ingenuity