Principles for secure implementation

Some of the slides and content are from Mike Hicks' Coursera course

Trusted computing bases

Every system has a TCB

- Your reference monitor
- Compiler
- OS
- CPU
- Memory
- Keyboard…..

Security requires the TCB be

- Correct
- Complete
- Secure

Security requires the TCB be

- Correct
- Complete
- Secure

Two principles behind a good TCB:

KISS Privilege Separation

KISS: Small TCB

- Keep the **TCB small** (and simple) to **reduce overall susceptibility to compromise**
  - The trusted computing base (TCB) comprises the system components that *must* work correctly to ensure security

- **Example**: **Operating system kernels**
  - Kernels enforce security policies, but are often millions of lines of code
    - Compromise in a device driver compromises security overall
  - Better: **Minimize size of kernel** to reduce trusted components
    - Device drivers moved outside of kernel in micro-kernel designs

Failure: Large TCB

Failure: Large TCB

- **Security software**
  is part of the TCB

Failure: Large TCB

- **Security software** is part of the TCB
- But as it grows in size and complexity, it becomes vulnerable itself, and can be bypassed

DARPA Additional security layers often create vulnerabilities…

October 2010 vulnerability watchlist

Vulnerability Title	Fix Avail?	Date Added
XXXXXXXXXXXX XXXXXXXXXXXX Local Privilege Escalation Vulnerability	No	8/25/2010
XXXXXXXXXXXX XXXXXXXXXXXX Denial of Service Vulnerability	Yes	8/24/2010
XXXXXXXXXXXX XXXXXXXXXXXX Buffer Overflow Vulnerability	No	8/20/2010
XXXXXXXXXXXX XXXXXXXXXXXX Sanitization Bypass Weakness	No	8/18/2010
XXXXXXXXXXXX XXXXXXXXXXXX Security Bypass Vulnerability	No	8/17/2010
XXXXXXXXXXXX XXXXXXXXXXXX Multiple Security Vulnerabilities	Yes	8/16/2010
XXXXXXXXXXXX XXXXXXXXXXXX Remote Code Execution Vulnerability	No	8/16/2010
XXXXXXXXXXXX XXXXXXXXXXXX Use-After-Free Memory Corruption Vulnerability	No	8/12/2010
XXXXXXXXXXXX XXXXXXXXXXXX Remote Code Execution Vulnerability	No	8/10/2010
XXXXXXXXXXXX XXXXXXXXXXXX Multiple Buffer Overflow Vulnerabilities	No	8
XXXXXXXXXXXX XXXXXXXXXXXX Stack Buffer Overflow Vulnerability	Yes	8
XXXXXXXXXXXX XXXXXXXXXXXX Security-Bypass Vulnerability	No	8
XXXXXXXXXXXX XXXXXXXXXXXX Multiple Security Vulnerabilities	No	8
XXXXXXXXXXXX XXXXXXXXXXXX Buffer Overflow Vulnerability	No	7/29/2010
XXXXXXXXXXXX XXXXXXXXXXXX Remote Privilege Escalation Vulnerability	No	7/28/2010
XXXXXXXXXXXX XXXXXXXXXXXX Cross Site Request Forgery Vulnerability	No	7/26/2010
XXXXXXXXXXXX XXXXXXXXXXXX Multiple Denial Of Service Vulnerabilities	No	7/22/2010

6 of the vulnerabilities are in security software

Color Code Key: Vendor Replied – Fix in development | Awaiting Vendor Reply/Confirmation | Awaiting CC/S/A use validation

http://www.darpa.mil/WorkArea/DownloadAsset.aspx?id=2147484449

TCB: Privilege Separation

Isolate privileged operations to as small a module as possible

- Don’t give a part of the system more privileges than it needs to do its job (“*need to know*”)
  - **Principle of least privilege**

TCB: Privilege Separation

Isolate privileged operations to as small a module as possible

- Don’t give a part of the system more privileges than it needs to do its job (“*need to know*”)
  - **Principle of least privilege**

- **Example**: Web server daemon

TCB: Privilege Separation

Isolate privileged operations to as small a module as possible

- Don’t give a part of the system more privileges than it needs to do its job (“*need to know*”)
  - **Principle of least privilege**

- **Example**: Web server daemon
  - Binding to port 80 requires root

TCB: Privilege Separation

Isolate privileged operations to as small a module as possible

- Don't give a part of the system more privileges than it needs to do its job ("*need to know*")
  - **Principle of least privilege**

- **Example**: Web server daemon
  - Binding to port 80 requires root
  - Don't want your whole web server running as root!

TCB: Privilege Separation

Isolate privileged operations to as small a module as possible

- Don’t give a part of the system more privileges than it needs to do its job (“*need to know*”)
  - **Principle of least privilege**

- **Example**: Web server daemon
  - Binding to port 80 requires root
  - Don’t want your whole web server running as root!

TCB: Privilege Separation

Isolate privileged operations to as small a module as possible

- Don’t give a part of the system more privileges than it needs to do its job (“*need to know*”)
  - **Principle of least privilege**

- **Example**: Web server daemon
  - Binding to port 80 requires root
  - Don’t want your whole web server running as root!

- **Example**: Email apps often drop you into an editor

TCB: Privilege Separation

Isolate privileged operations to as small a module as possible

- Don't give a part of the system more privileges than it needs to do its job ("*need to know*")
  - **Principle of least privilege**

- **Example**: Web server daemon
  - Binding to port 80 requires root
  - Don't want your whole web server running as root!

- **Example**: Email apps often drop you into an editor
  - vi, emacs

TCB: Privilege Separation

Isolate privileged operations to as small a module as possible

- Don't give a part of the system more privileges than it needs to do its job ("*need to know*")
  - **Principle of least privilege**

- **Example**: Web server daemon
  - Binding to port 80 requires root
  - Don't want your whole web server running as root!

- **Example**: Email apps often drop you into an editor
  - vi, emacs
  - But these editors often permit dropping you into a shell

Lesson: Trust is Transitive

- **If you trust something, you trust what it trusts**
  - *This trust can be misplaced*

- **Previous e-mail client example**
  - Mailer delegates to an arbitrary editor
  - The editor permits running arbitrary code
  - Hence the mailer permits running arbitrary code

SecComp

SecComp

- Linux system call enabled since 2.6.12 (2005)

SecComp

- Linux system call enabled since 2.6.12 (2005)
  - Affected process can subsequently **only perform `read`, `write`, `exit`, and `sigreturn` system calls**
    - No support for `open` call: Can only use already-open file descriptors

SecComp

- Linux system call enabled since 2.6.12 (2005)
  - Affected process can subsequently **only perform `read`, `write`, `exit`, and `sigreturn` system calls**
    - No support for `open` call: Can only use already-open file descriptors
  - **Isolates a process by limiting possible interactions**

SecComp

- Linux system call enabled since 2.6.12 (2005)
  - Affected process can subsequently **only perform `read`, `write`, `exit`, and `sigreturn` system calls**
    - No support for `open` call: Can only use already-open file descriptors
  - **Isolates a process by limiting possible interactions**

- Follow-on work produced **seccomp-bpf**

SecComp

- Linux system call enabled since 2.6.12 (2005)
  - Affected process can subsequently **only perform `read`, `write`, `exit`, and `sigreturn` system calls**
    - No support for `open` call: Can only use already-open file descriptors
  - **Isolates a process by limiting possible interactions**

- Follow-on work produced **seccomp-bpf**
  - **Limit process to policy-specific set of system calls**, subject to a policy handled by the kernel
    - Policy akin to *Berkeley Packet Filters (BPF)*

SecComp

- Linux system call enabled since 2.6.12 (2005)
  - Affected process can subsequently **only perform `read`, `write`, `exit`, and `sigreturn` system calls**
    - No support for `open` call: Can only use already-open file descriptors
  - **Isolates a process by limiting possible interactions**

- Follow-on work produced **seccomp-bpf**
  - **Limit process to policy-specific set of system calls**, subject to a policy handled by the kernel
    - Policy akin to *Berkeley Packet Filters (BPF)*
  - Used by *Chrome*, *OpenSSH*, *vsftpd*, and others

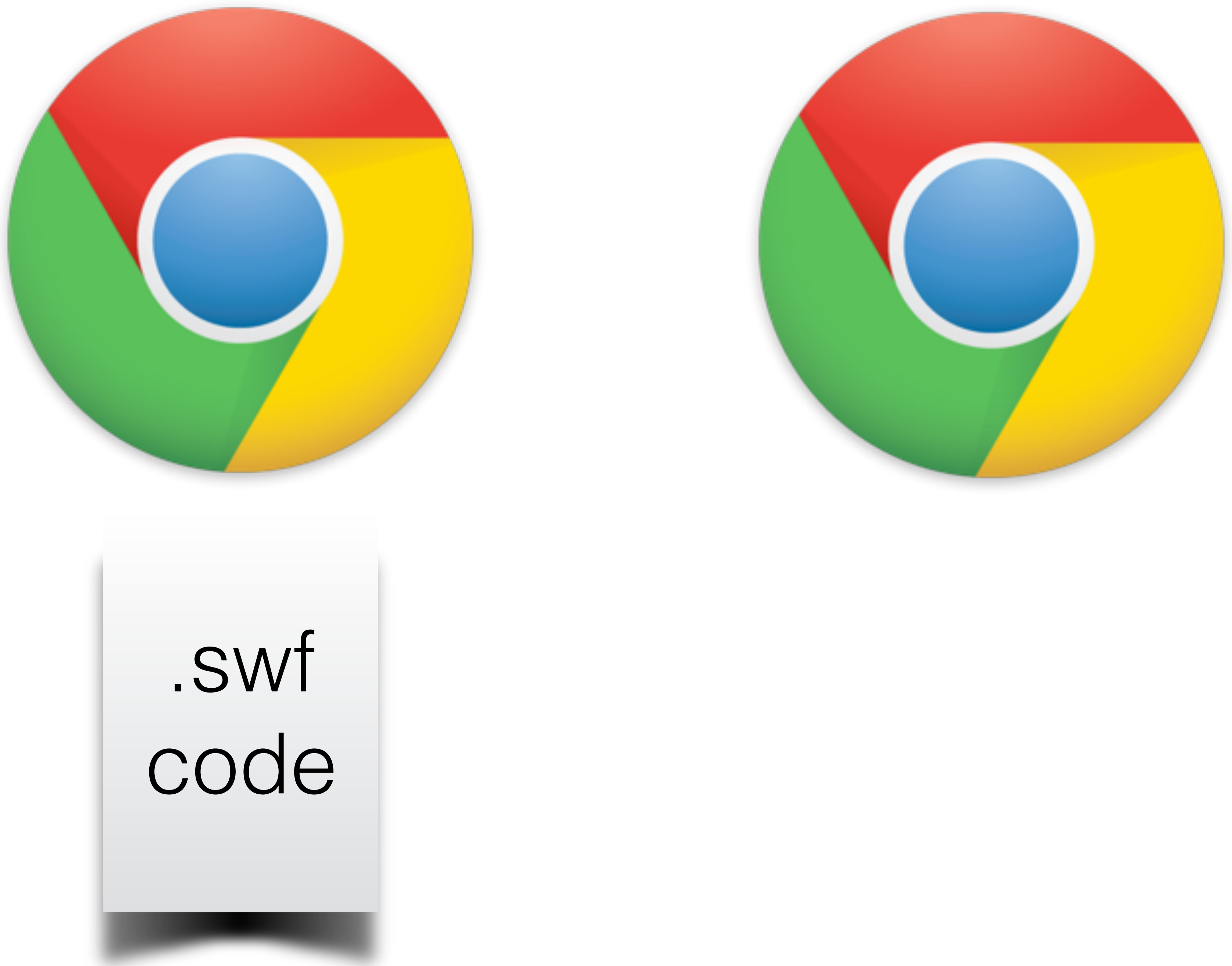
Idea: Isolate Flash Player

Idea: Isolate Flash Player

- Receive .swf code, save it

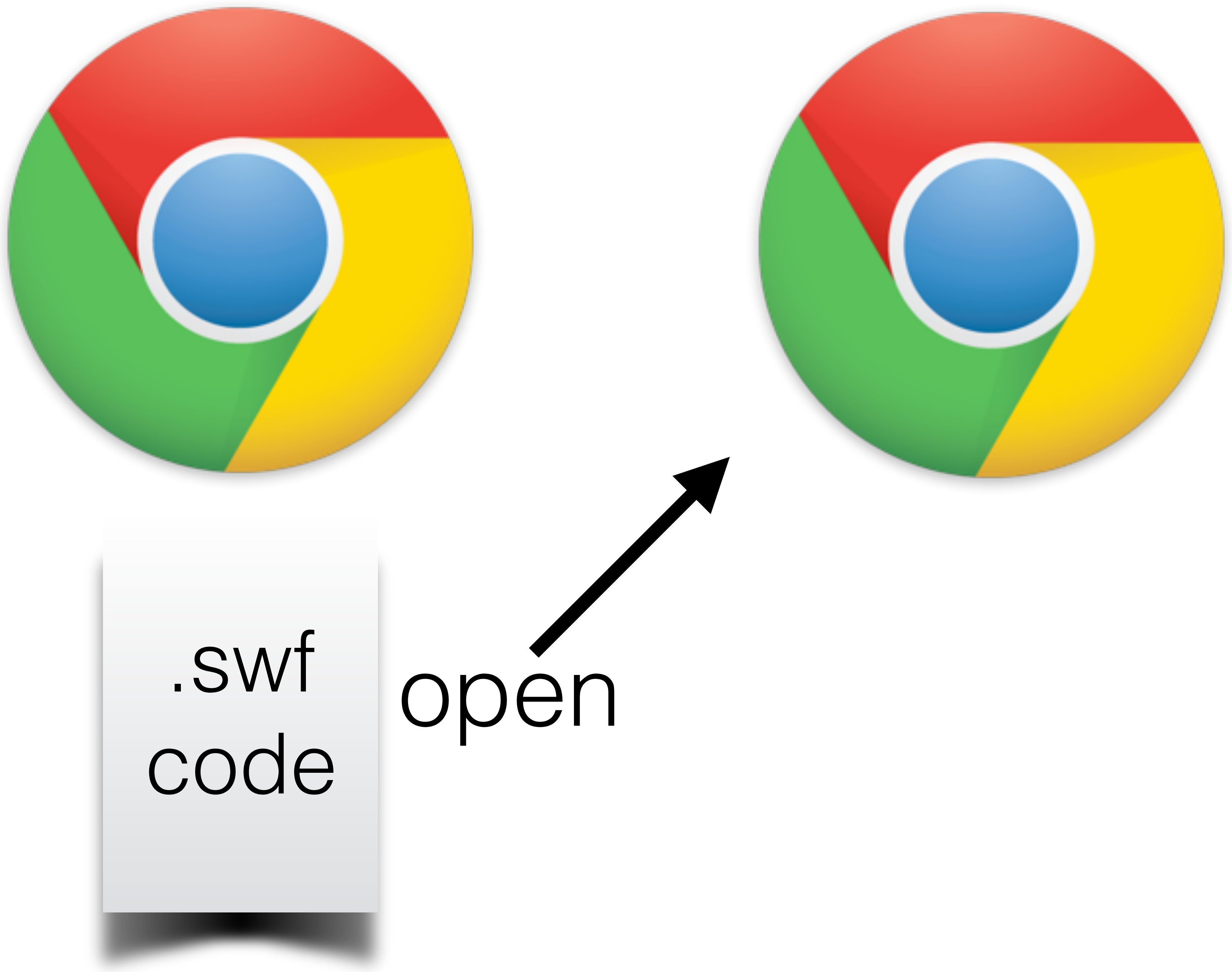
Idea: Isolate Flash Player

- Receive .swf code, save it
- Call **`fork`** to create a new process



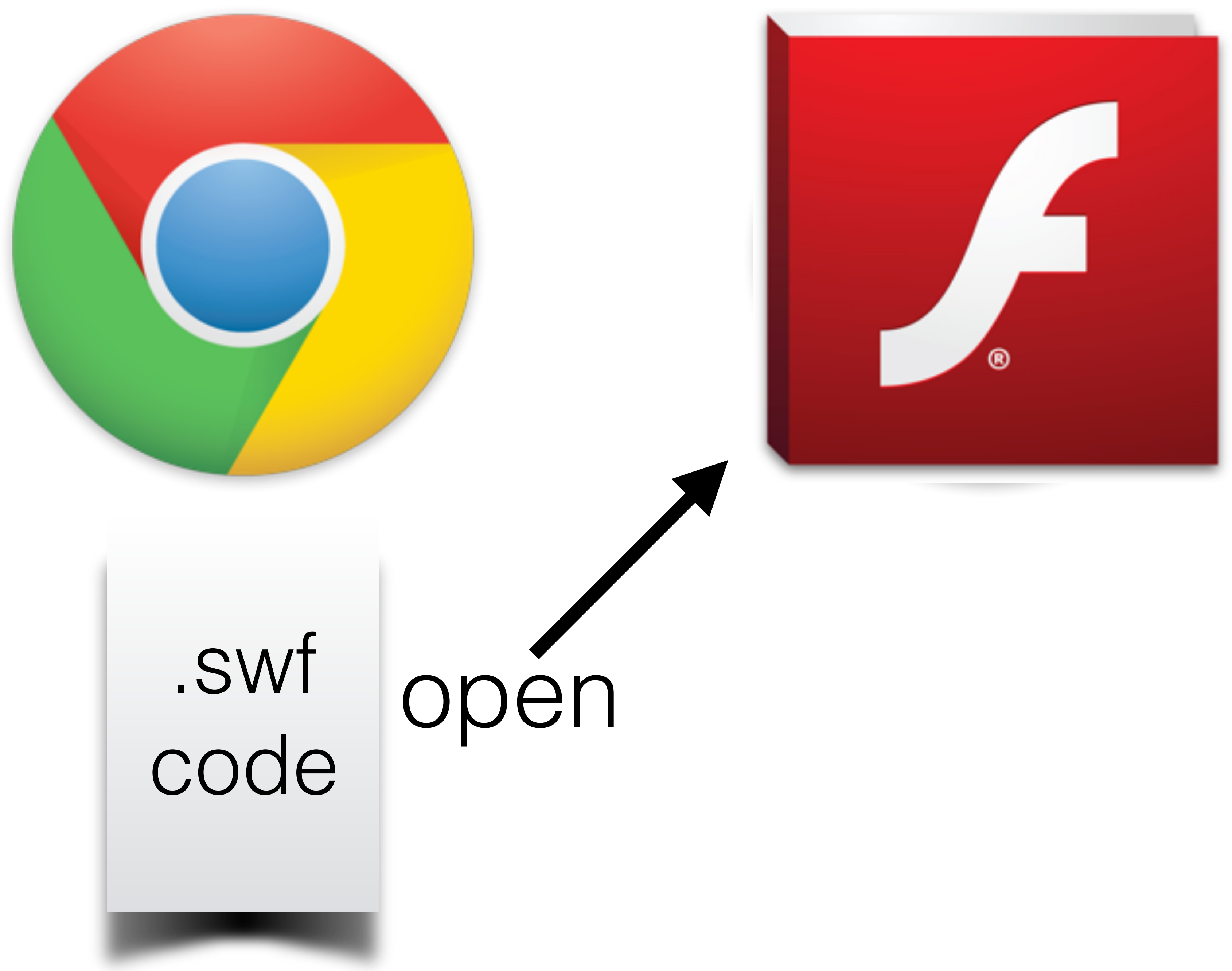
Idea: Isolate Flash Player

- Receive .swf code, save it
- Call `fork` to create a new process
- In the new process, open the file



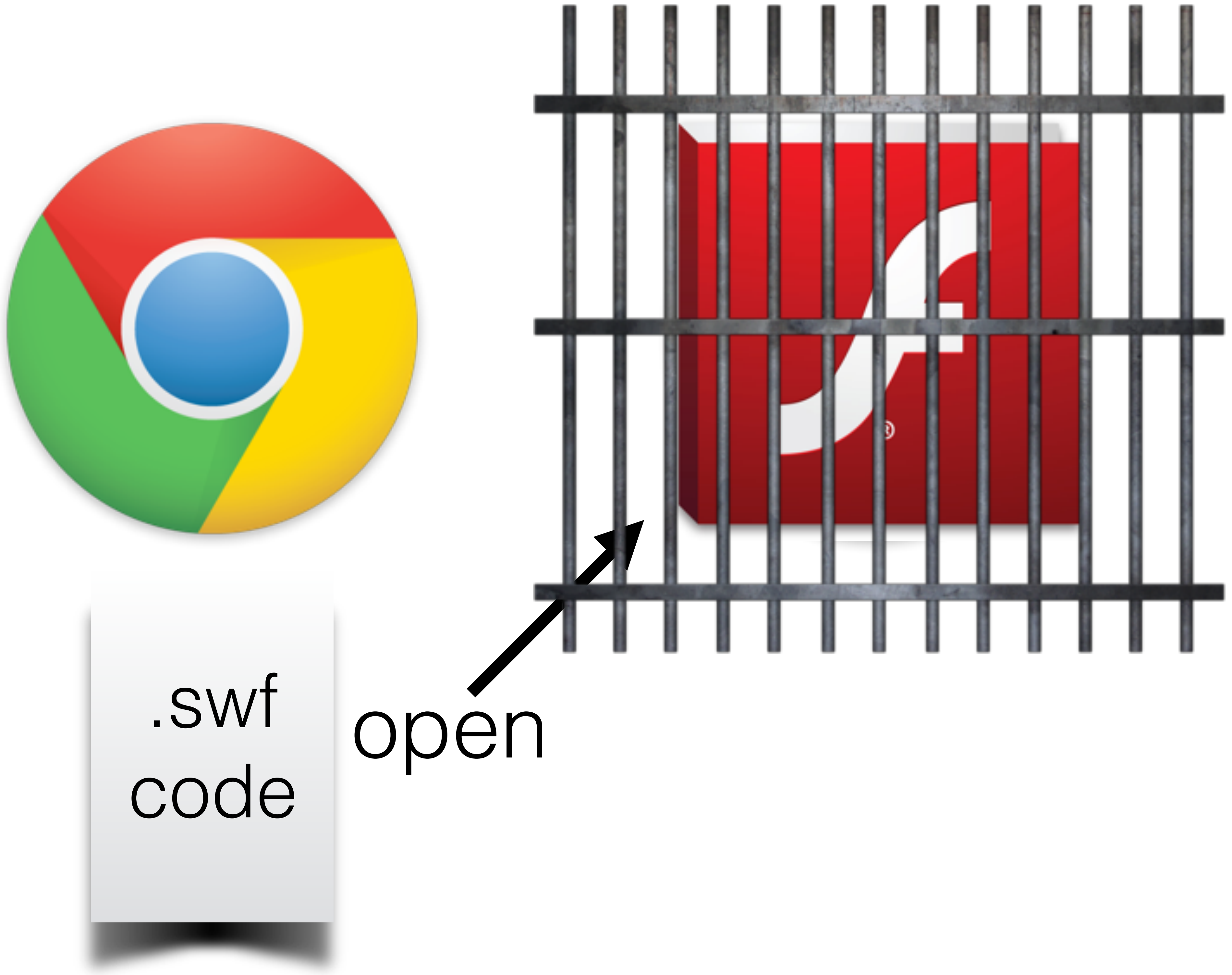
Idea: Isolate Flash Player

- Receive .swf code, save it
- Call `fork` to create a new process
- In the new process, open the file
- Call `exec` to run Flash player



Idea: Isolate Flash Player

- Receive .swf code, save it
- Call `fork` to create a new process
- In the new process, open the file
- Call `exec` to run Flash player
- Call `seccomp-bpf` to compartmentalize



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

VSFTPD Threat model

- **Clients untrusted, until authenticated**

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)

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

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)

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**

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:

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**

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;
};
```

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

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)`)

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`

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`

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

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

```c
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);
}
```

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

```c
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;
}
```

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

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

```c
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

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

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

```c
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

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

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

```c
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

```c
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**

Defense: Secure Stdcalls

- Common problem: **error handling**
  - Libraries **assume** that **arguments are well-formed**

Defense: Secure Stdcalls

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

Defense: Secure Stdcalls

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

- Example: `malloc()`

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

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;
}
```

```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

Defense: Minimal Privilege

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

Defense: Minimal Privilege

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

- **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

Defense: Minimal Privilege

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

- **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

Defense: Minimal Privilege

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

- **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

Defense: Minimal Privilege

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

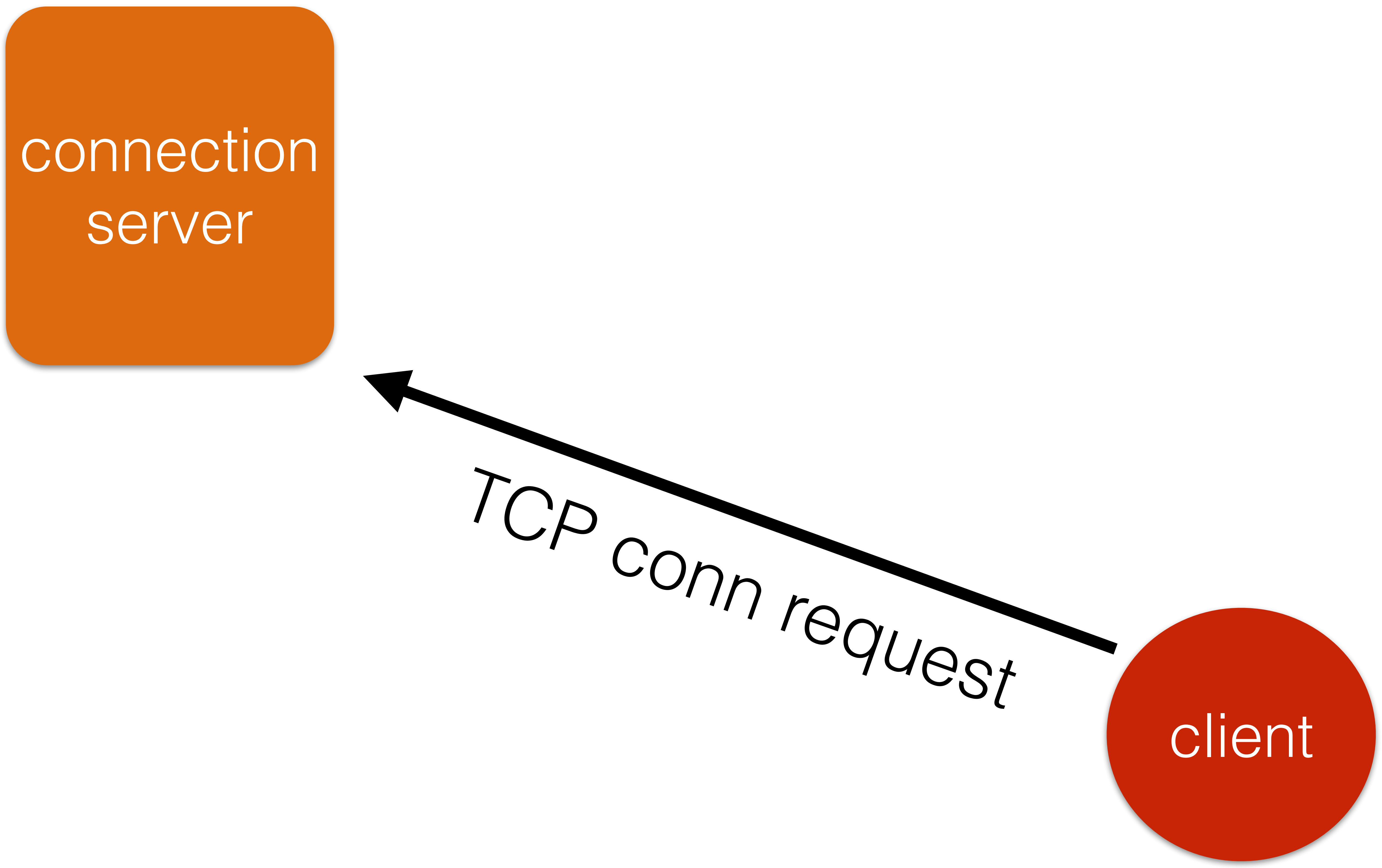
small trusted computing base

- **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

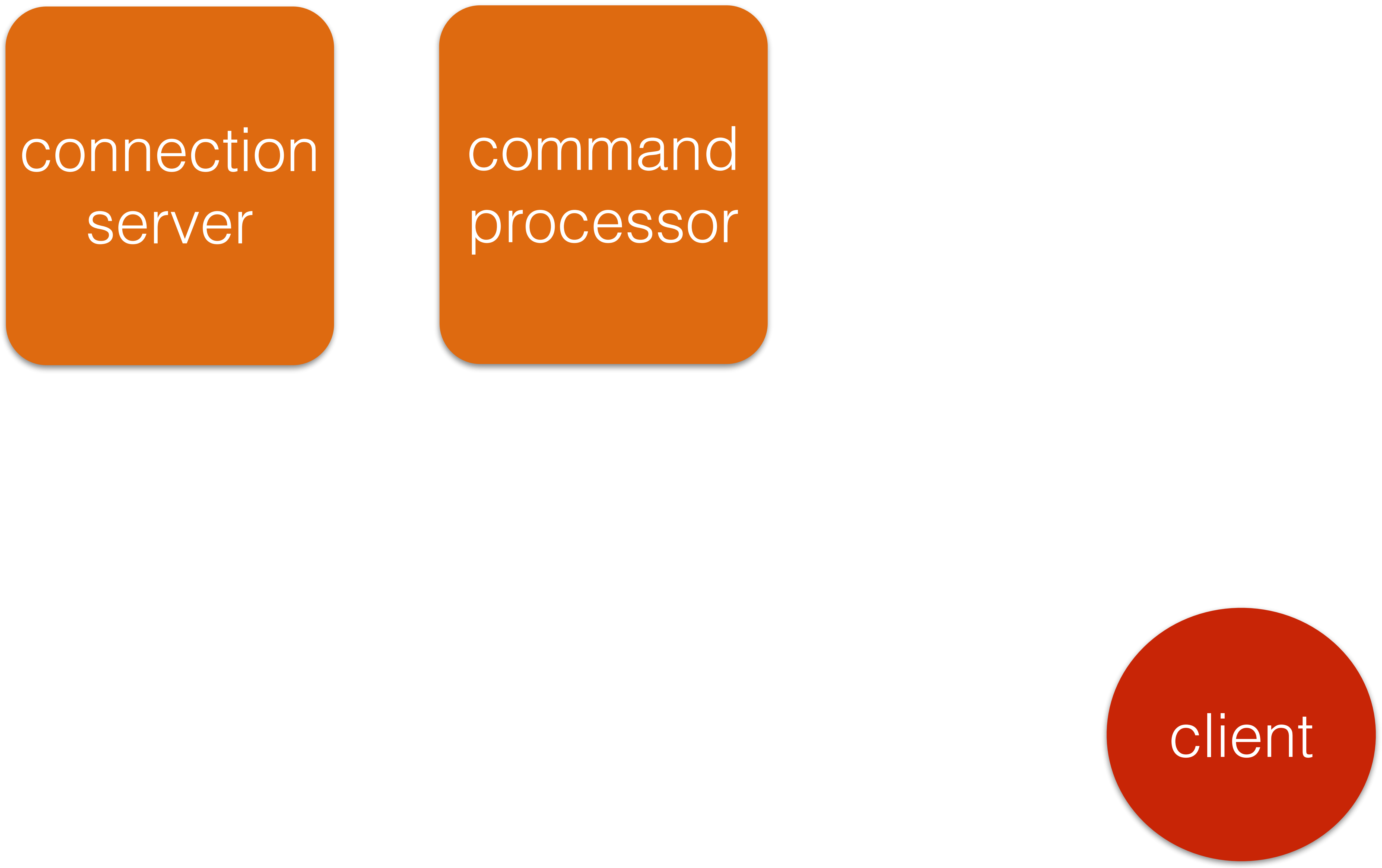
principle of least privilege

Connection Establishment

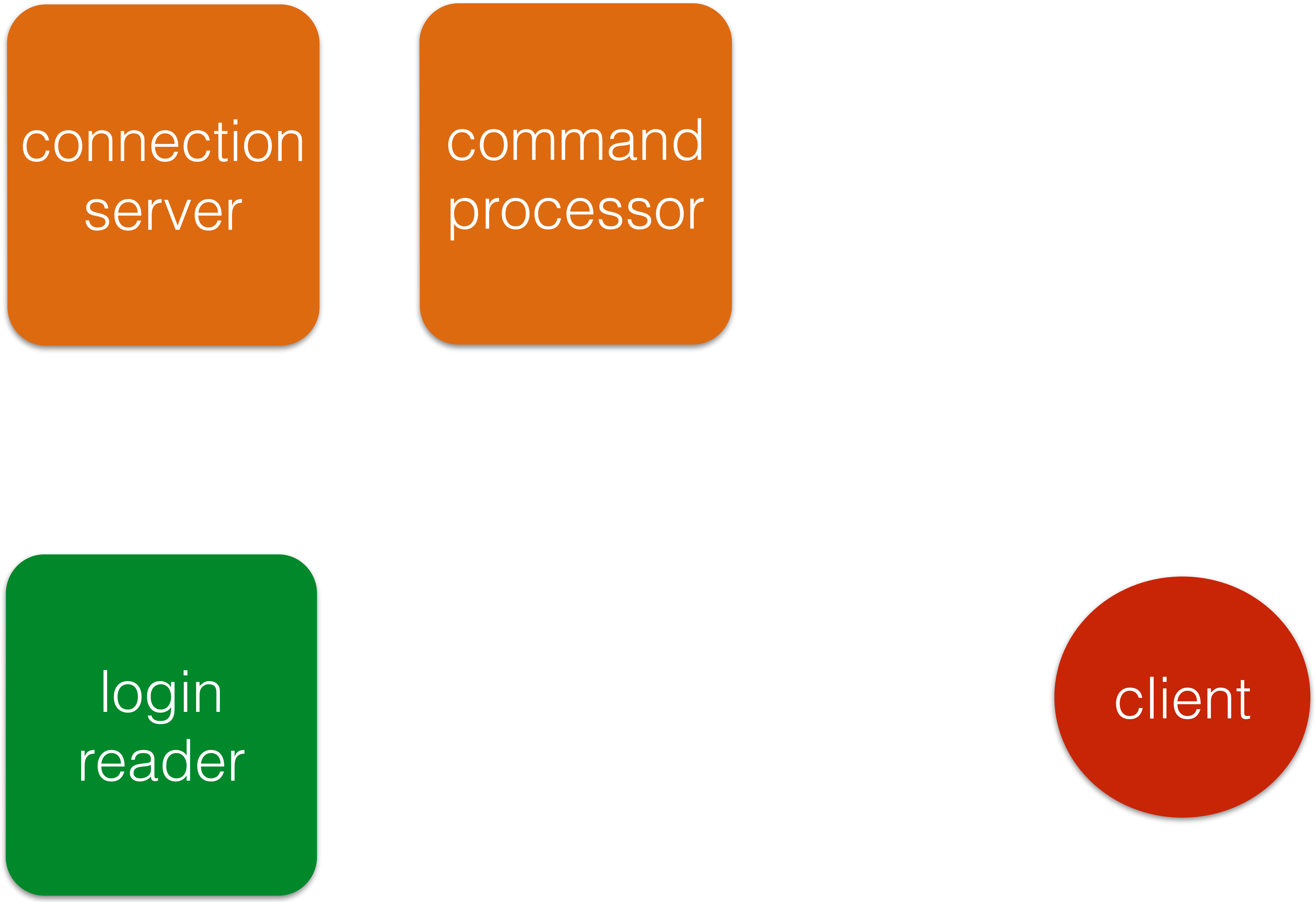
Connection Establishment



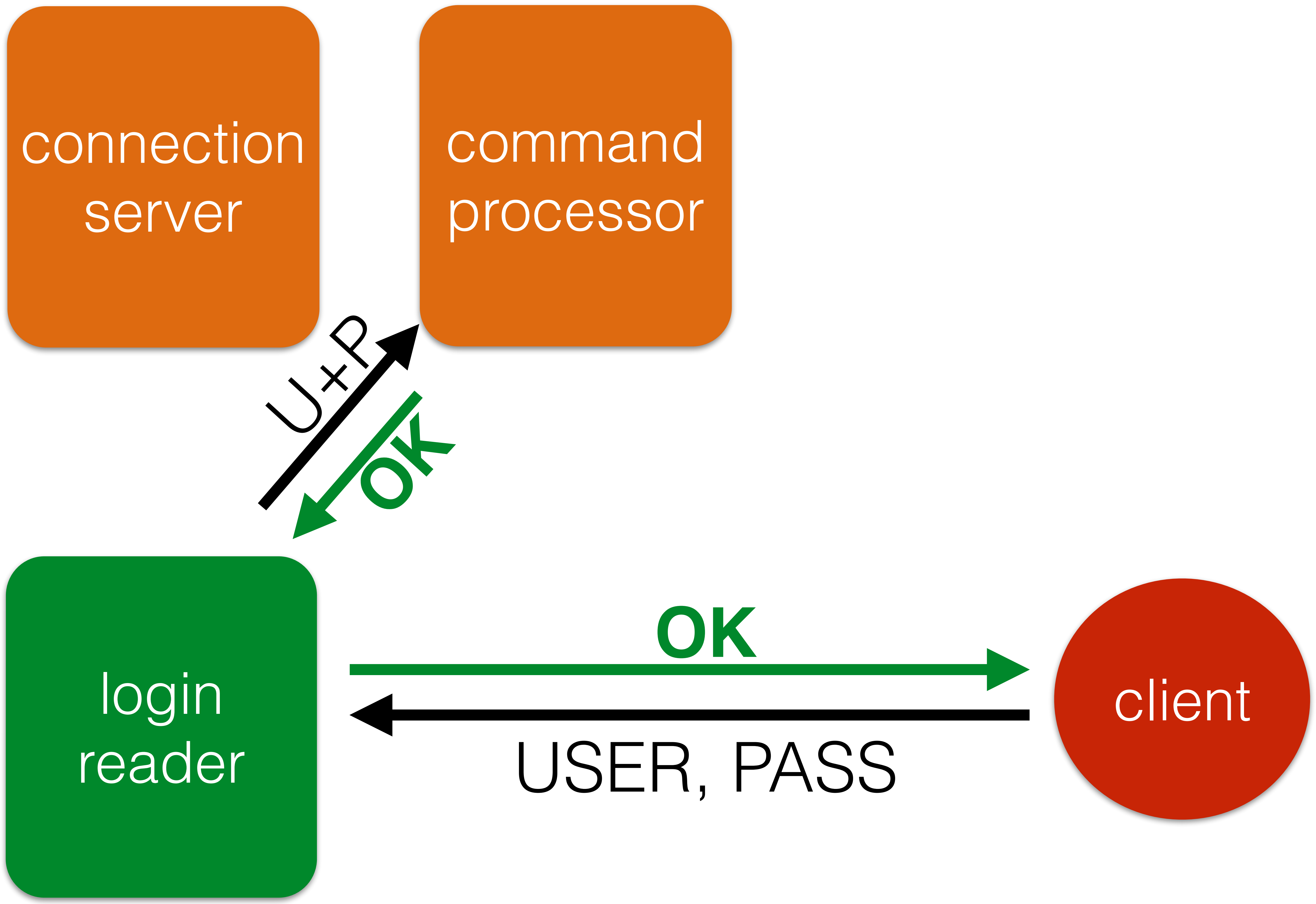
Connection Establishment



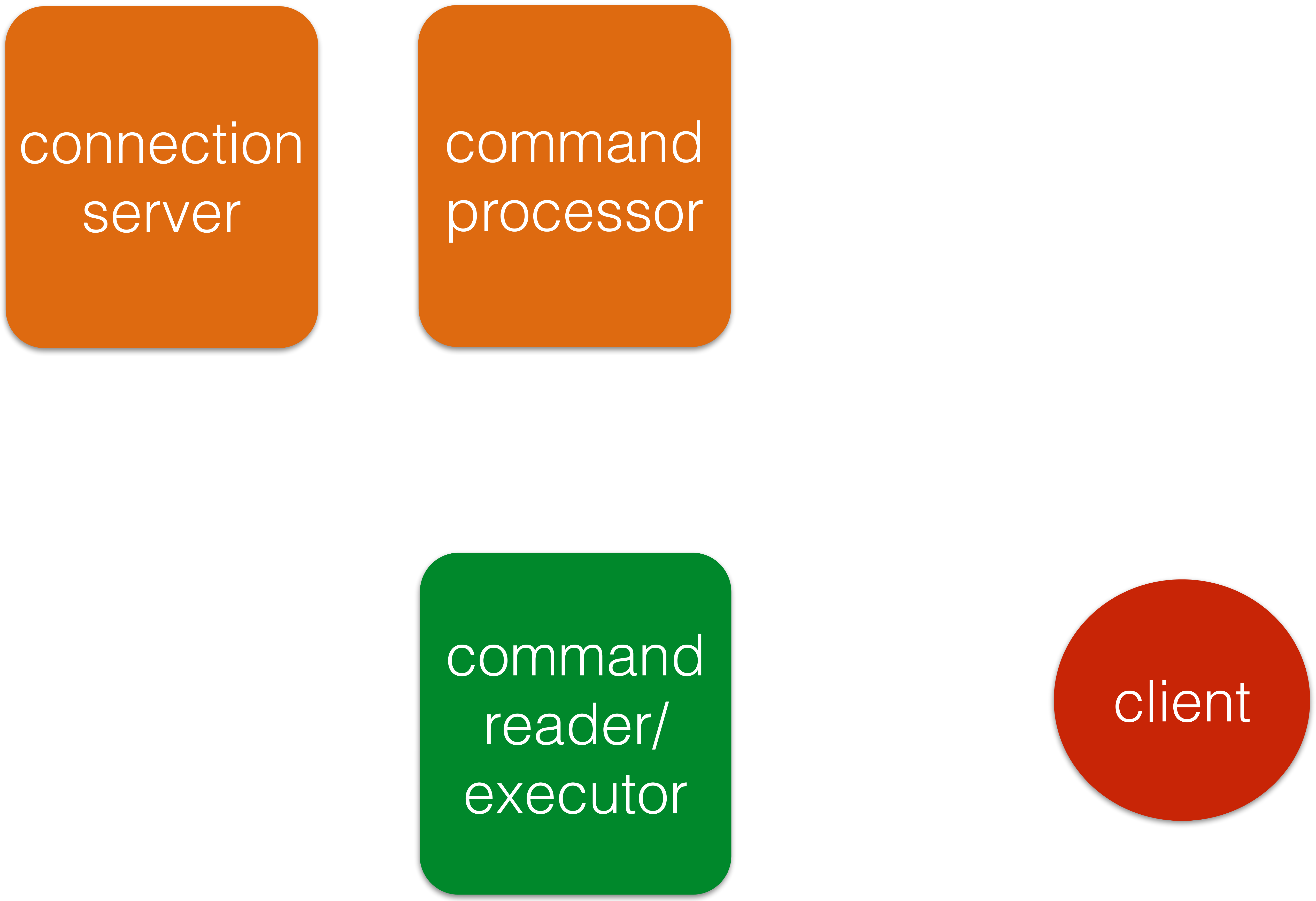
Connection Establishment



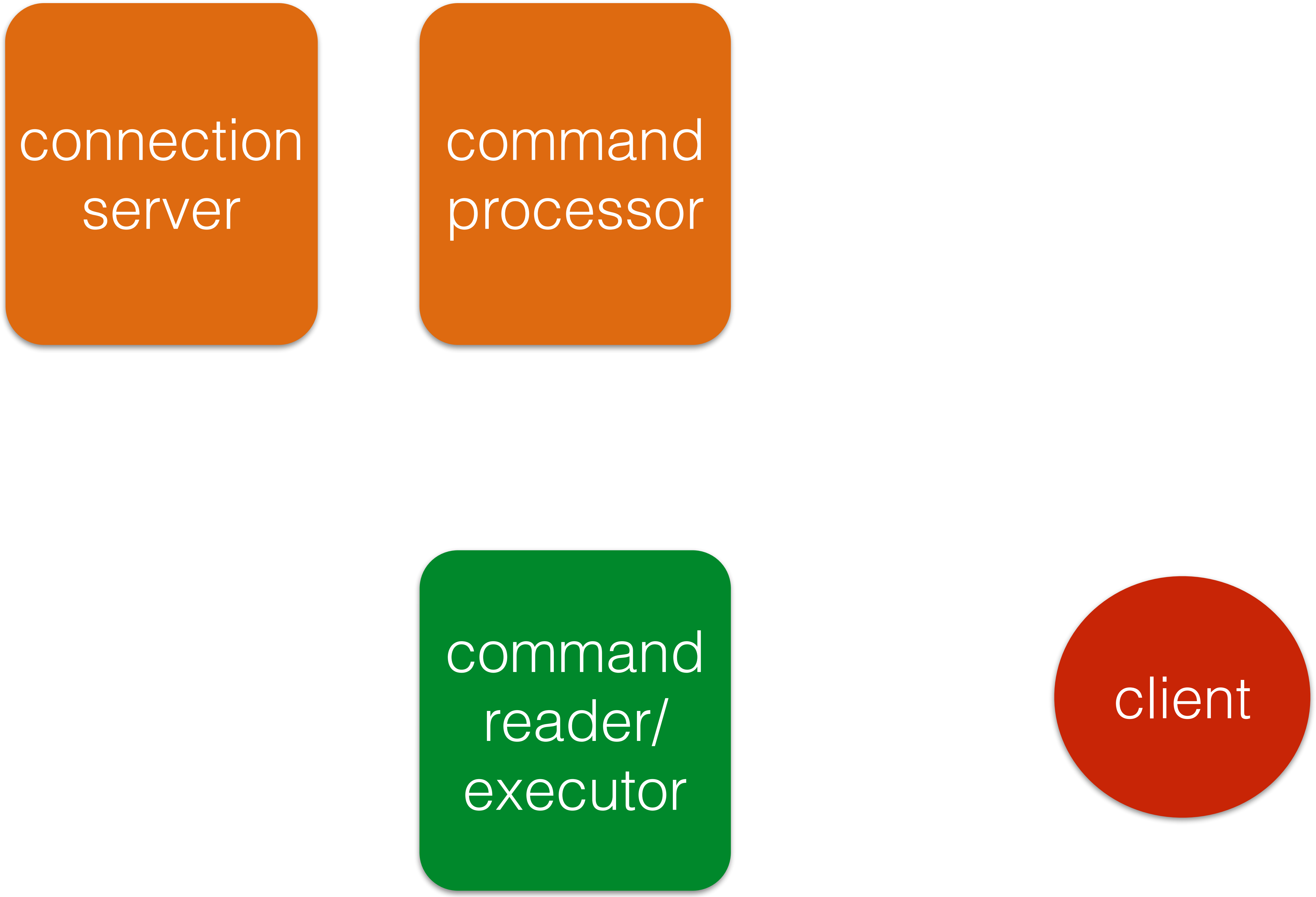
Connection Establishment



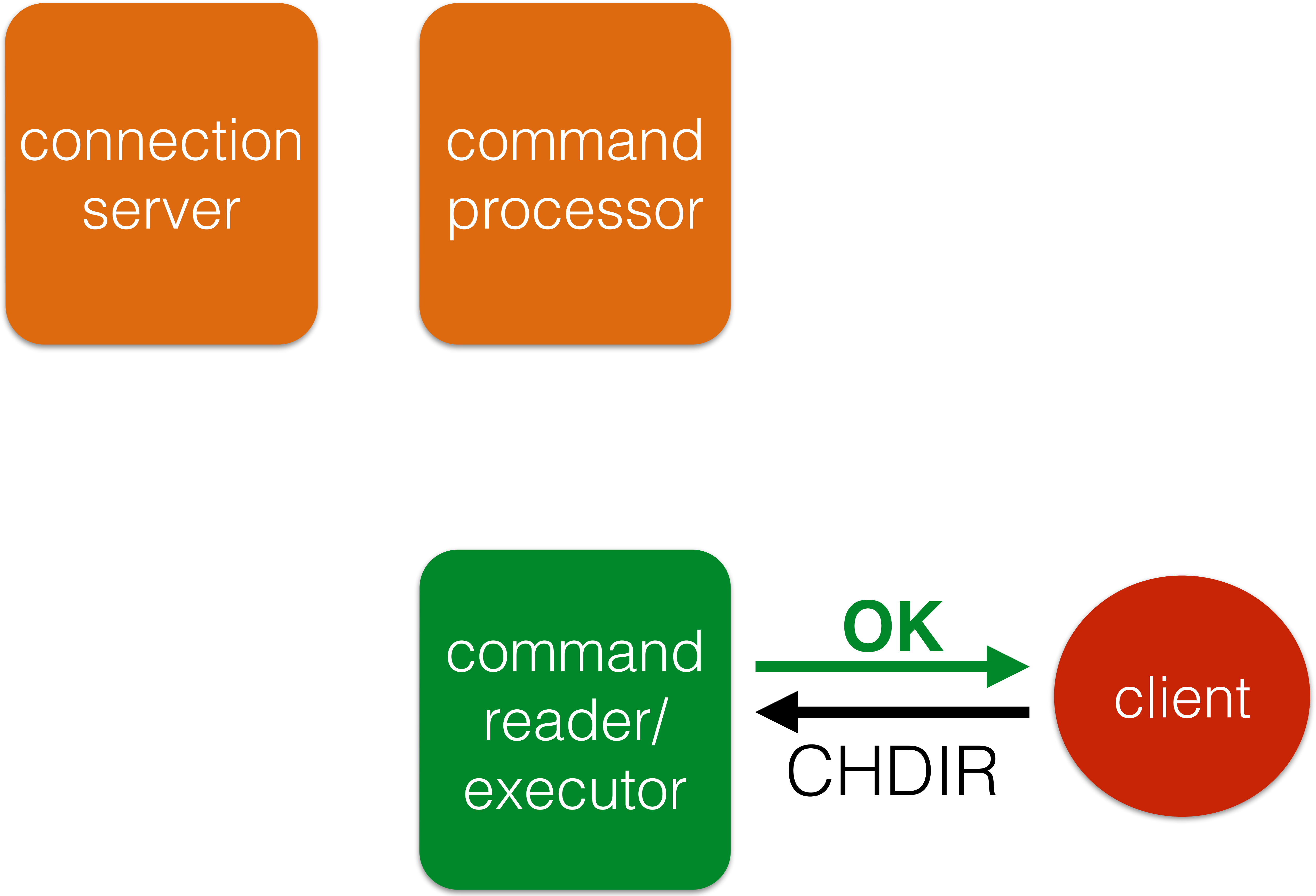
Connection Establishment



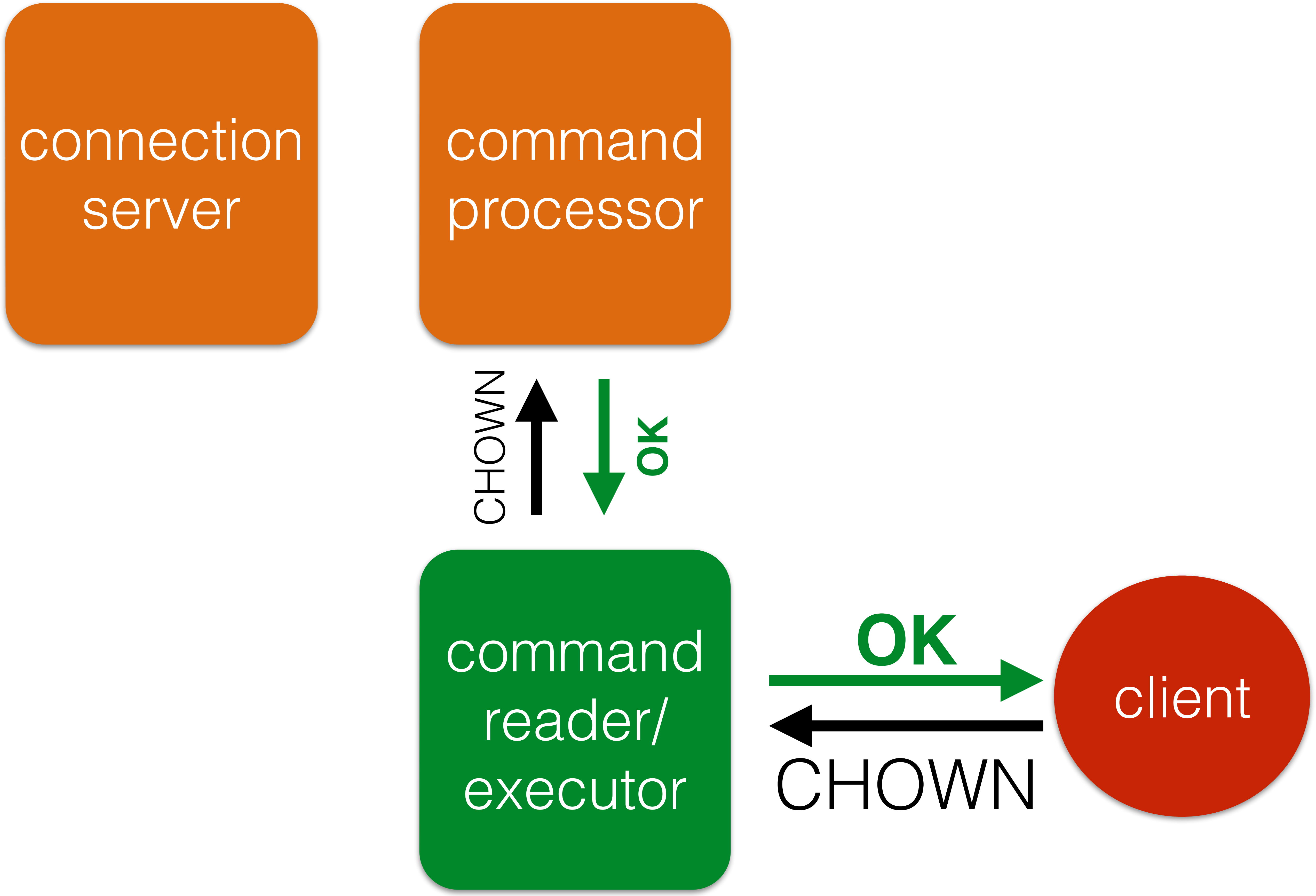
Performing Commands



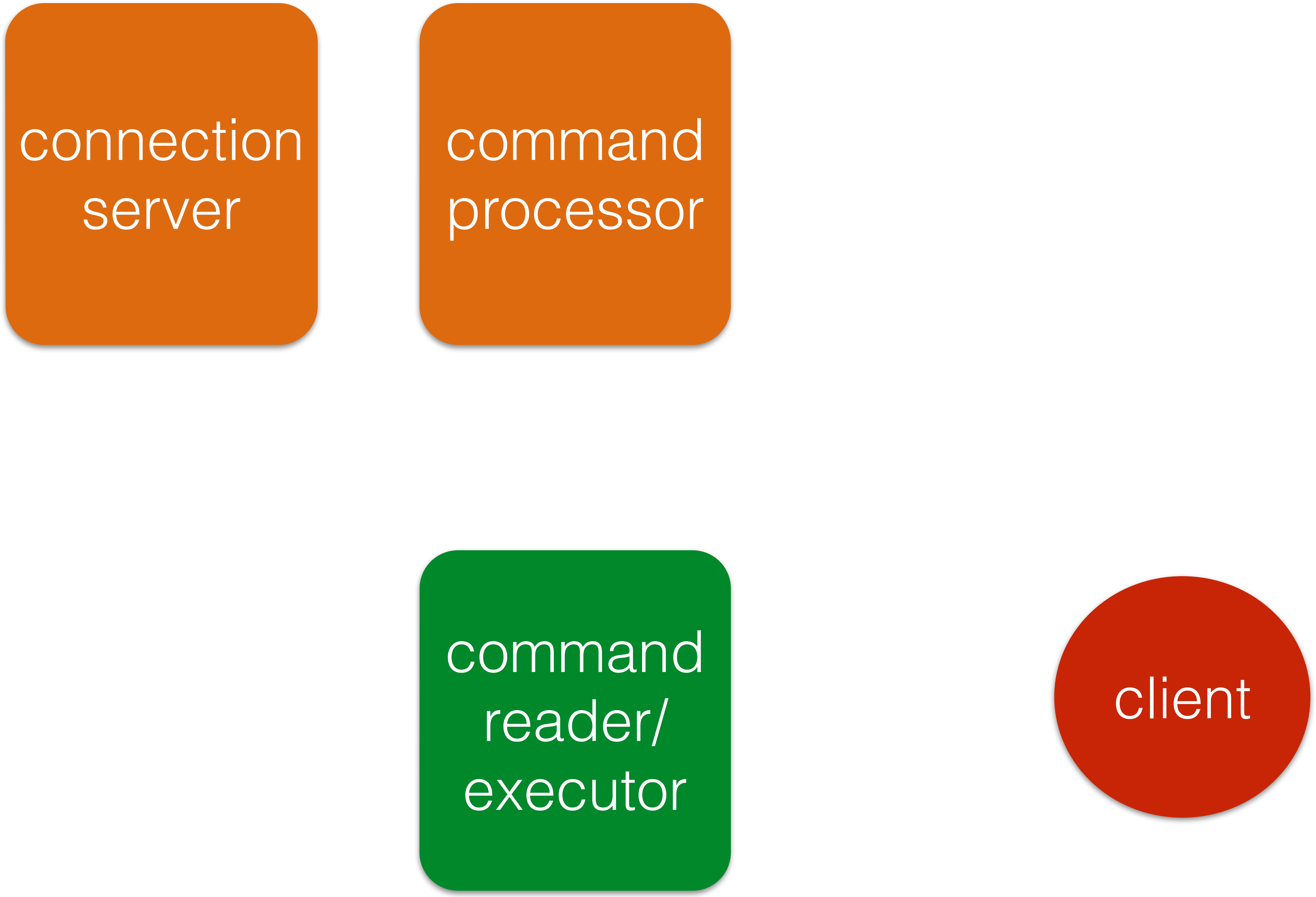
Performing Commands



Performing Commands

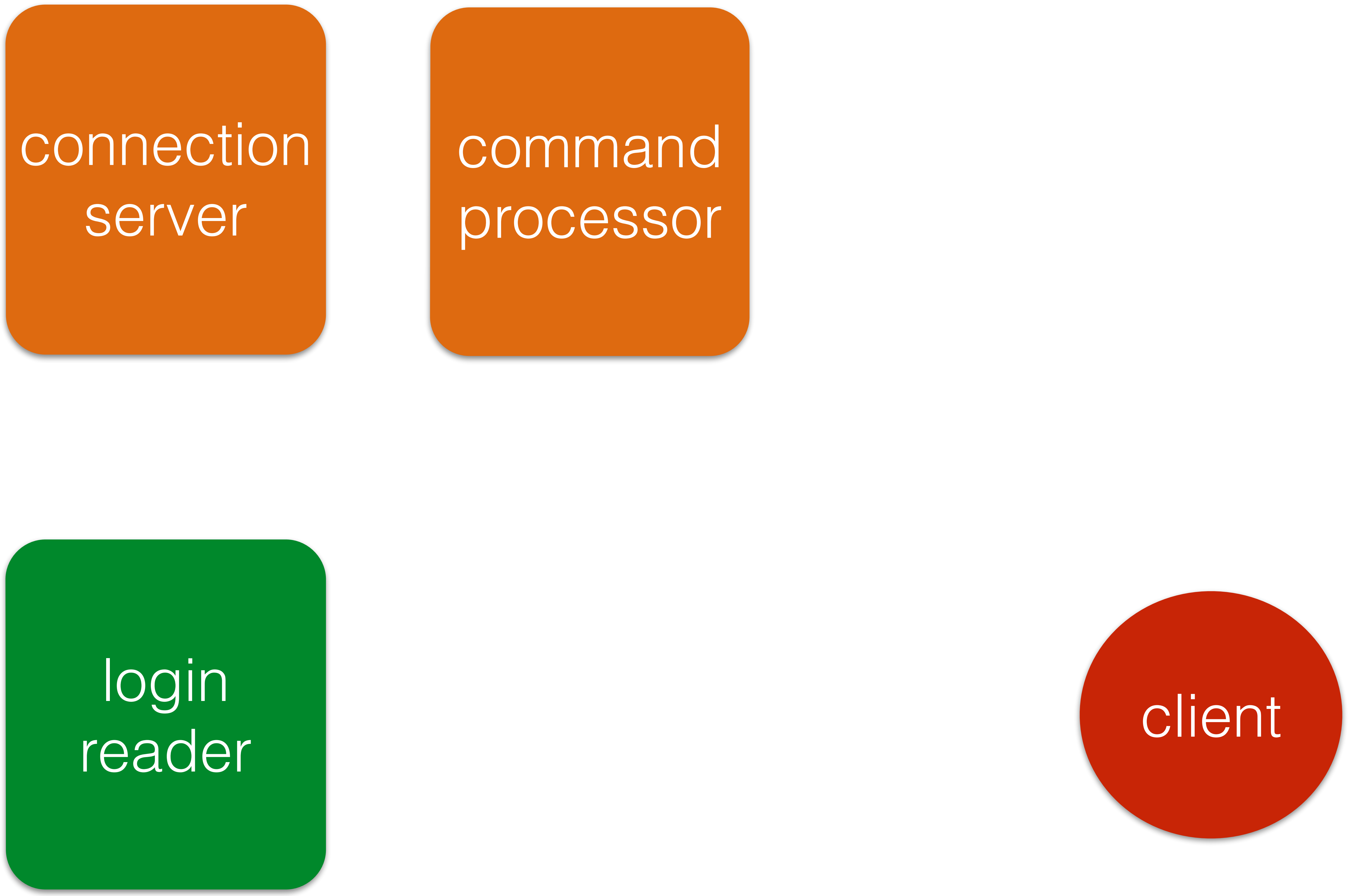


Logging out

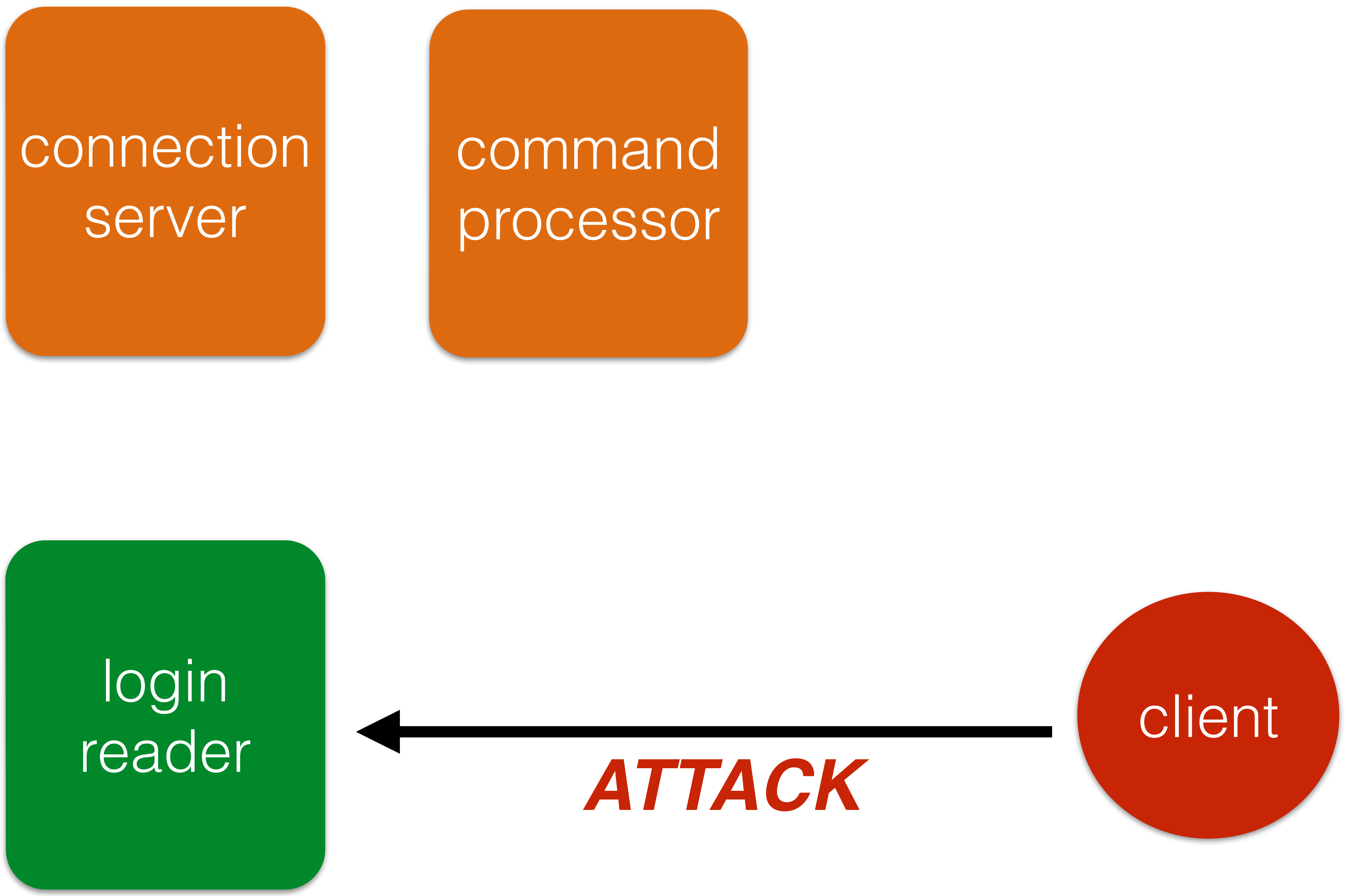

connection server
command processor
command reader/ executor
client

Logging out

Attack: Login

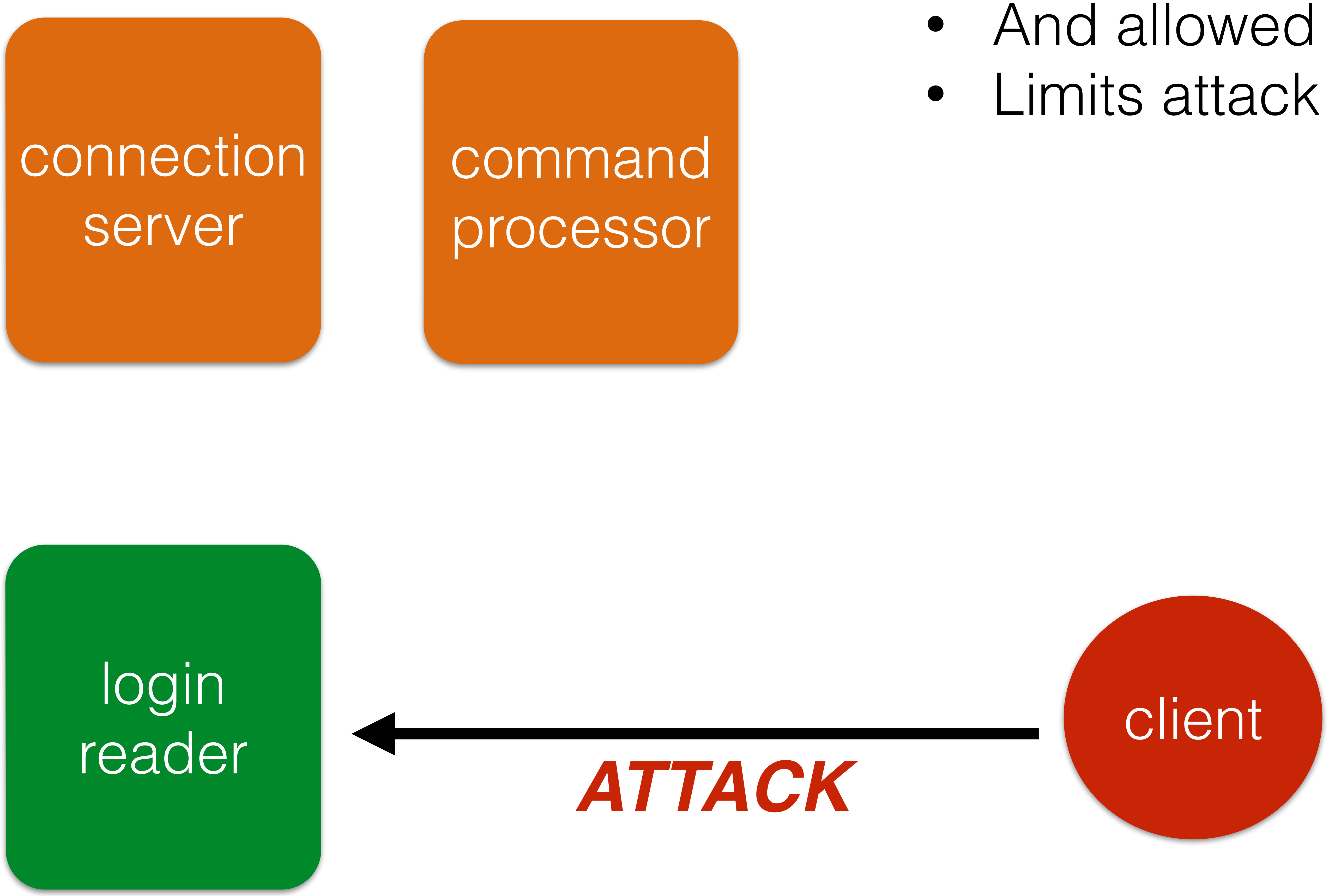


Attack: Login

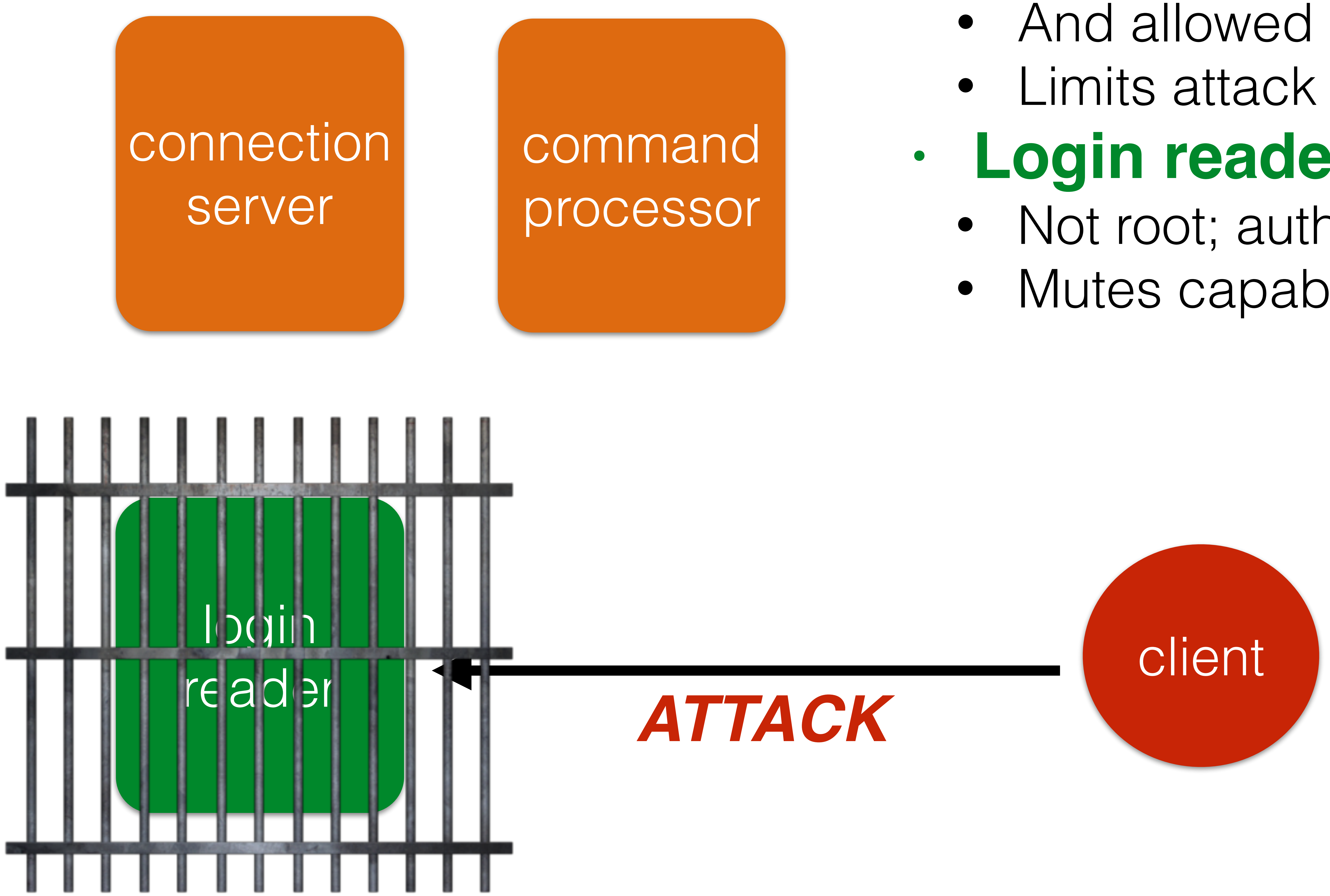


Attack: Login

- **Login reader white-lists input**
  - And allowed input very limited
  - Limits attack surface

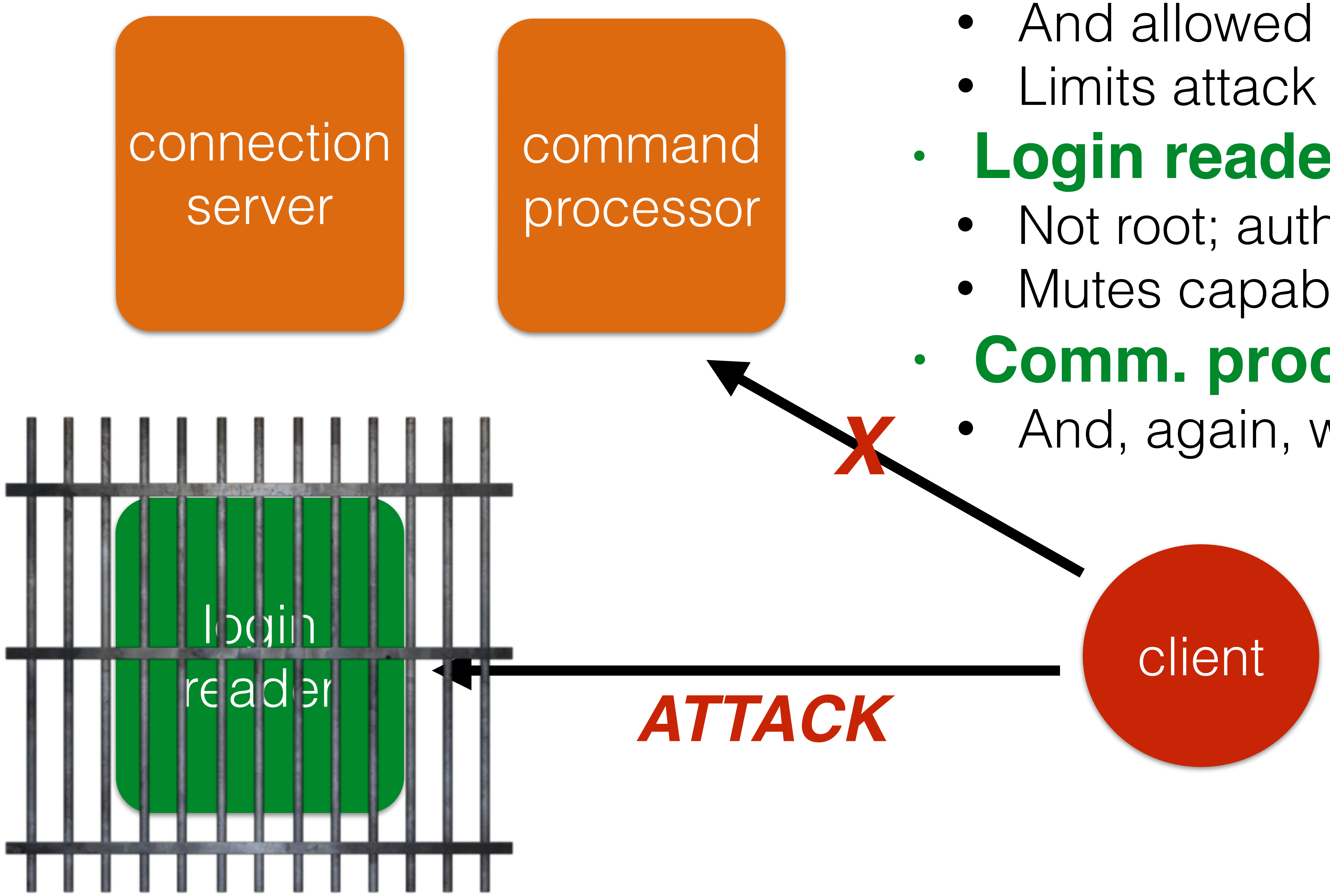


Attack: Login



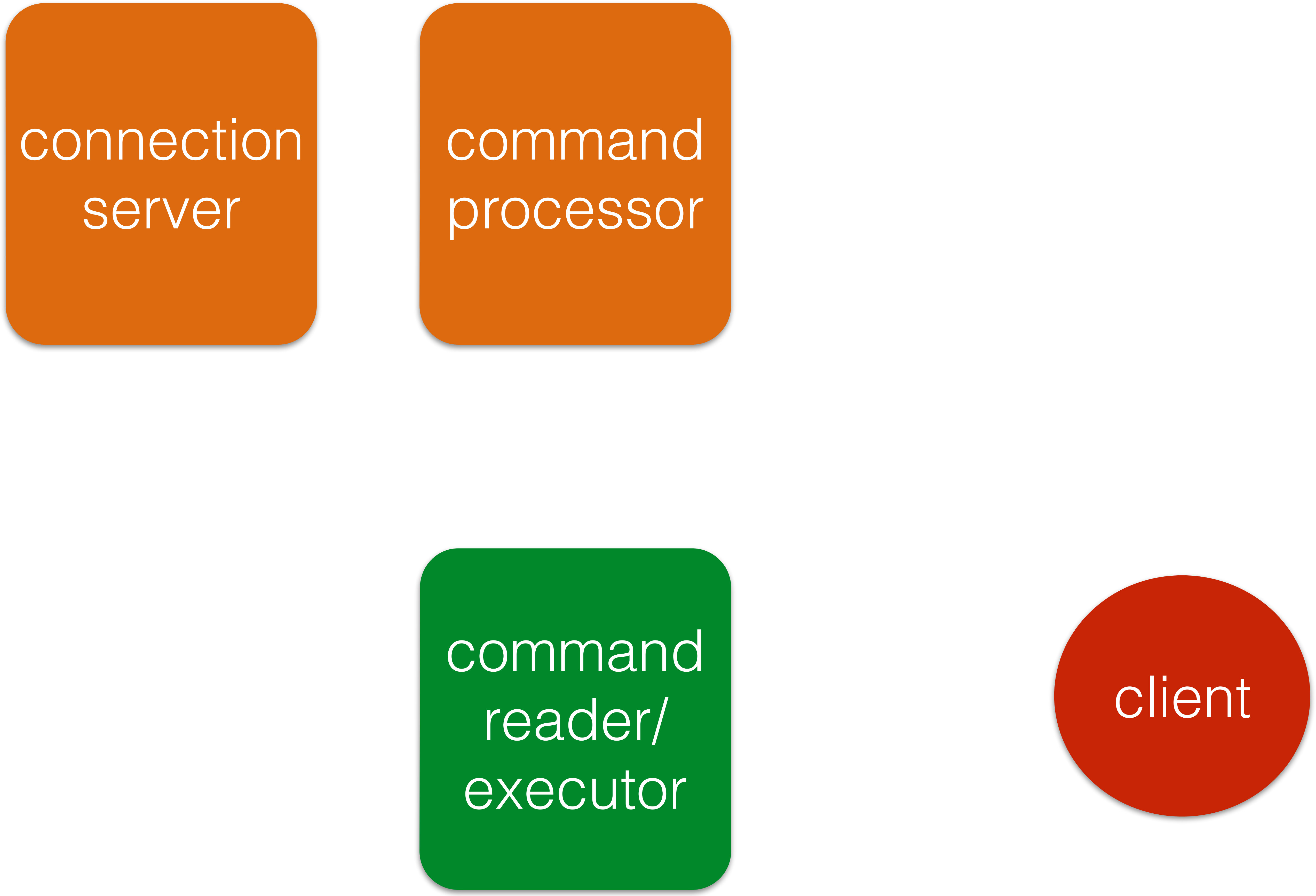
- **Login reader white-lists input**
  - And allowed input very limited
  - Limits attack surface
- **Login reader has limited privilege**
  - Not root; authentication in separate process
  - Mutes capabilities of injected code

Attack: Login

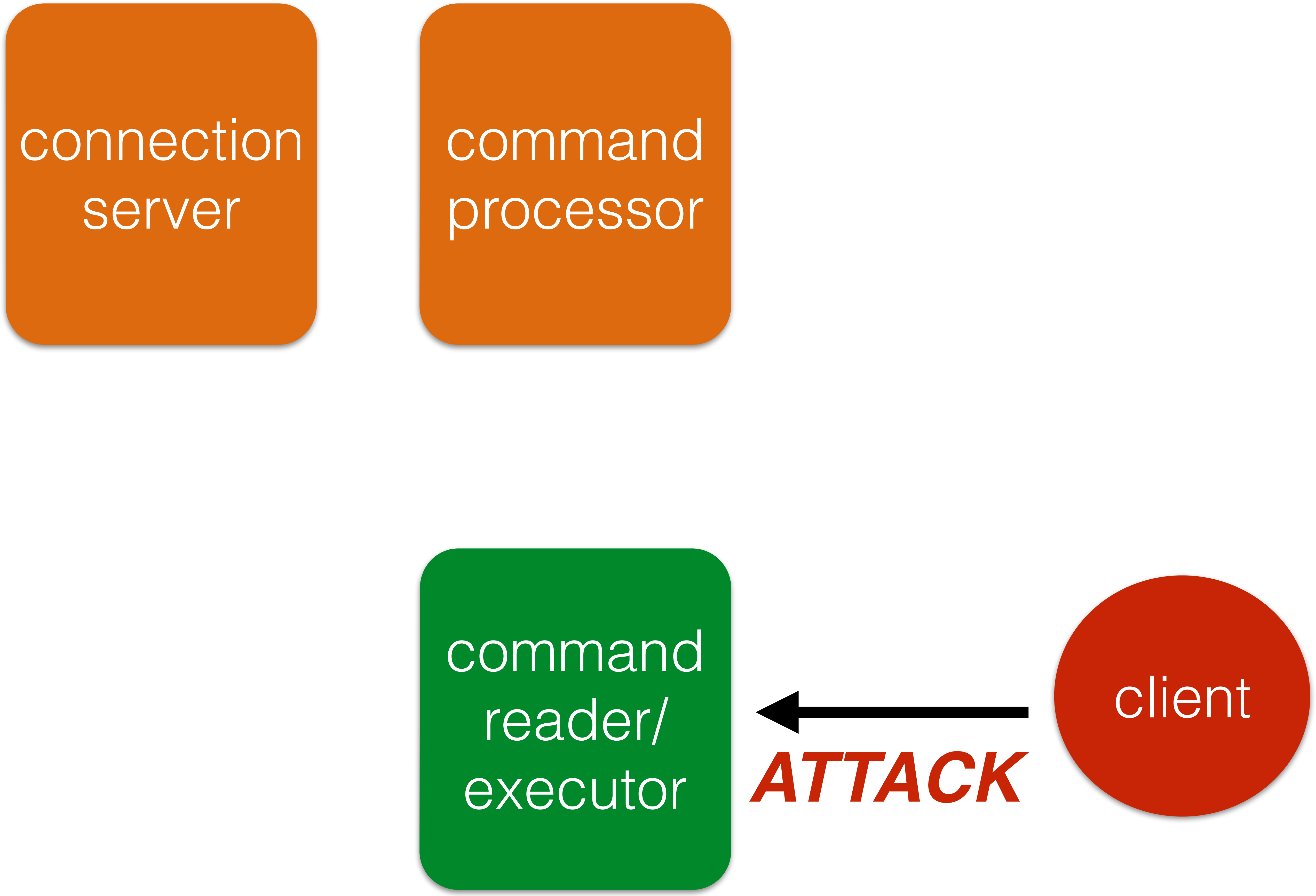


- **Login reader white-lists input**
  - And allowed input very limited
  - Limits attack surface
- **Login reader has limited privilege**
  - Not root; authentication in separate process
  - Mutes capabilities of injected code
- **Comm. proc. only talks to reader**
  - And, again, white-lists its limited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

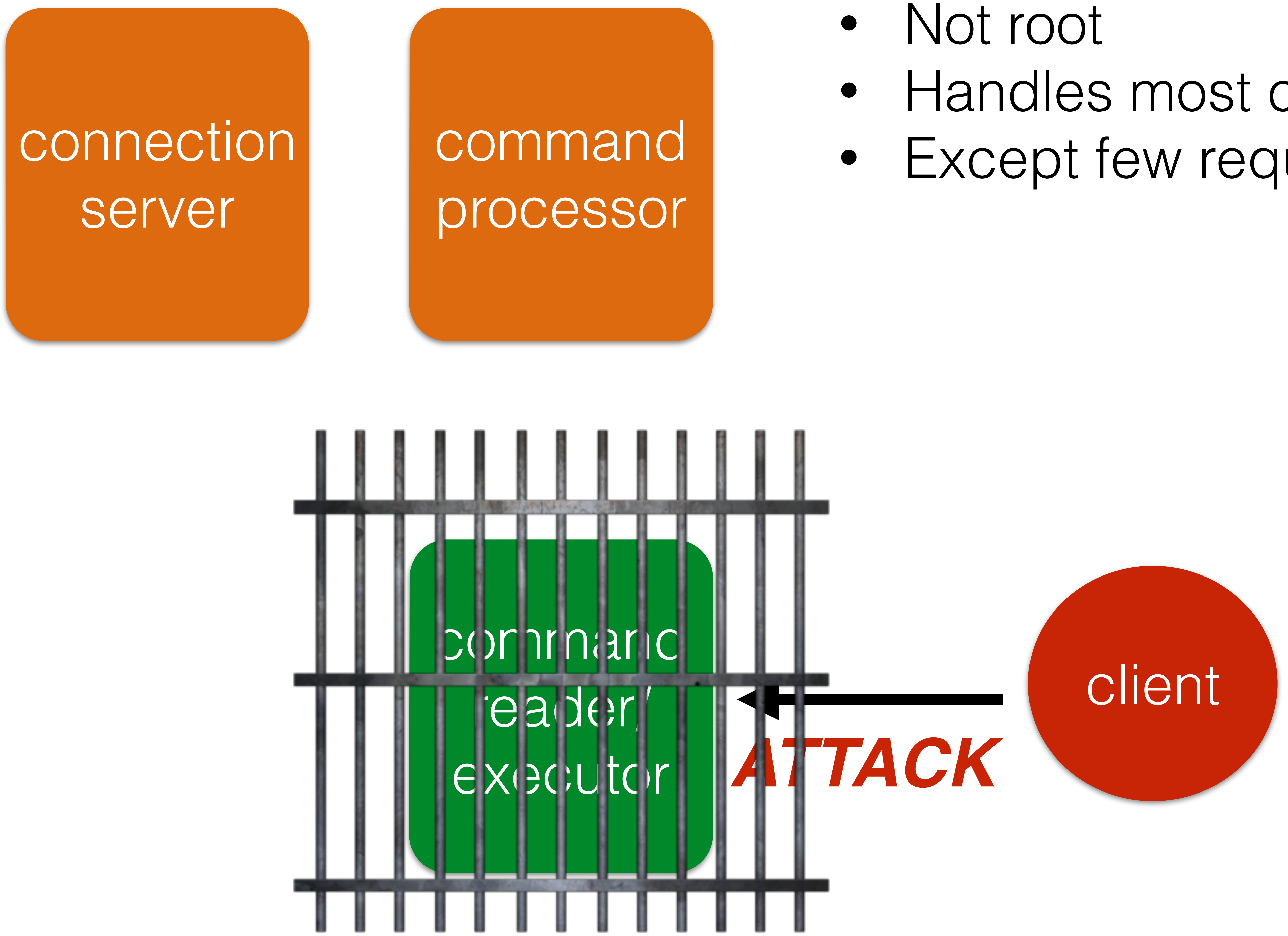
Attack: Commands



Attack: Commands

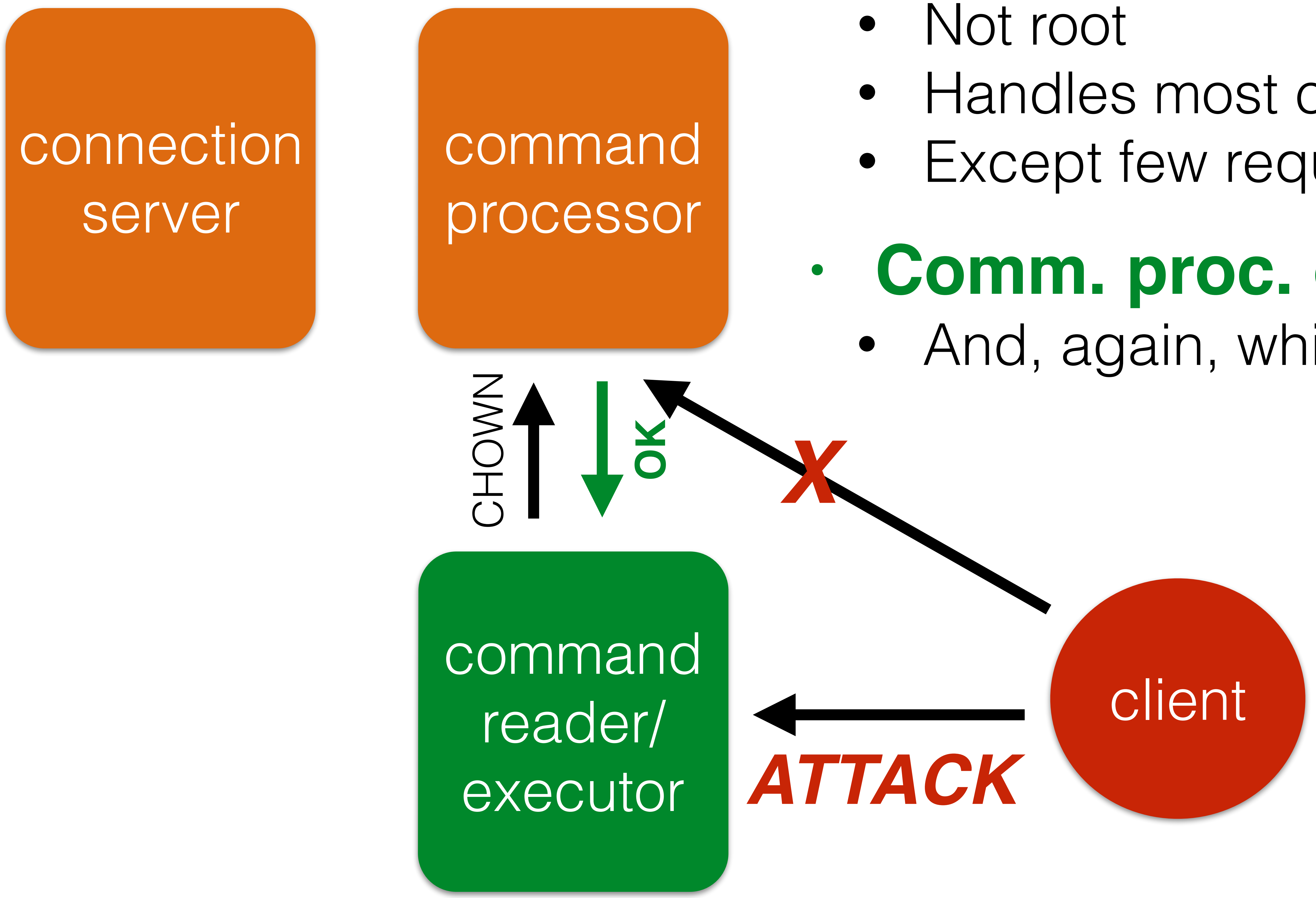


Attack: Commands



- **Command reader sandboxed**
  - Not root
  - Handles most commands
  - Except few requiring privilege

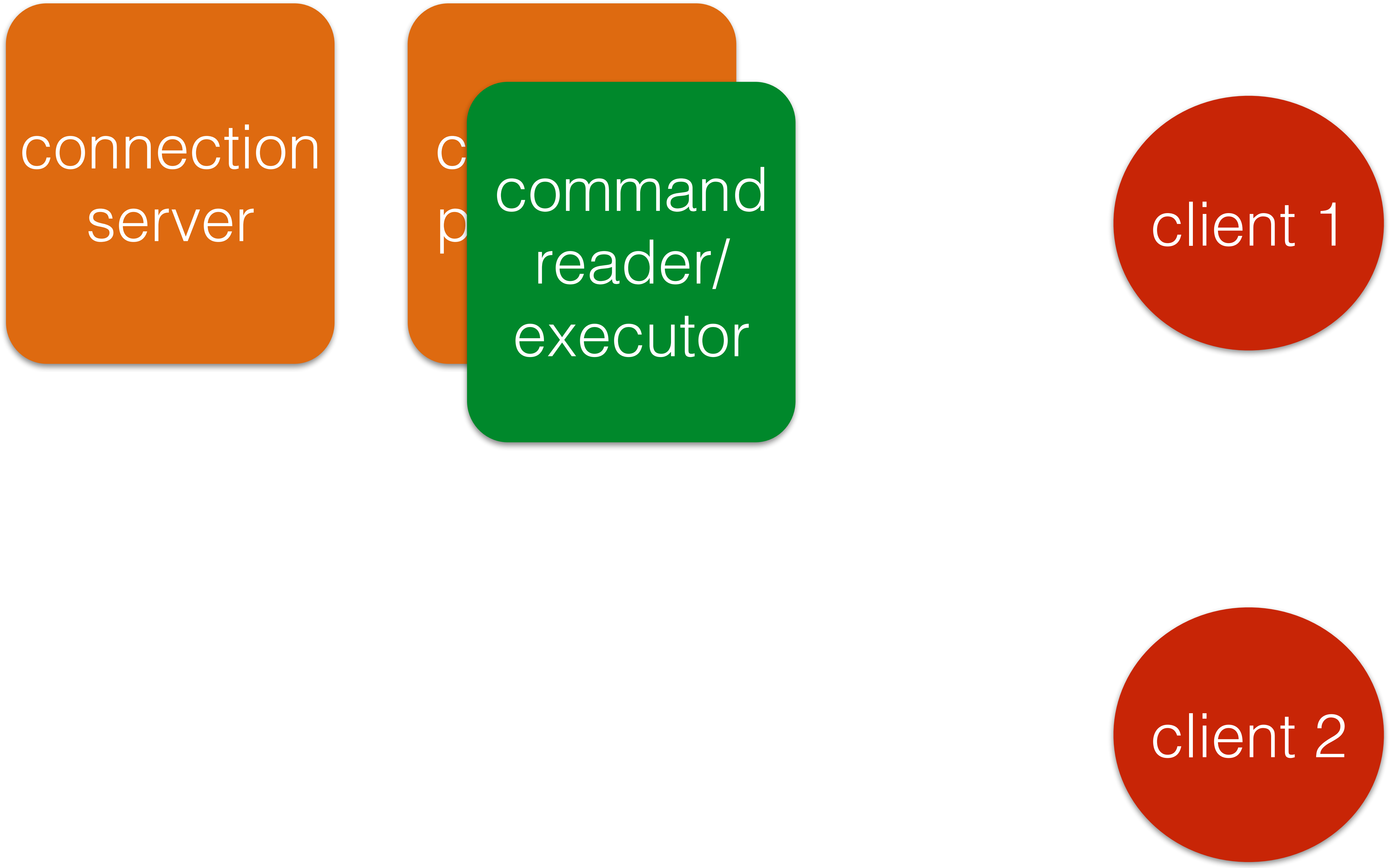
Attack: Commands



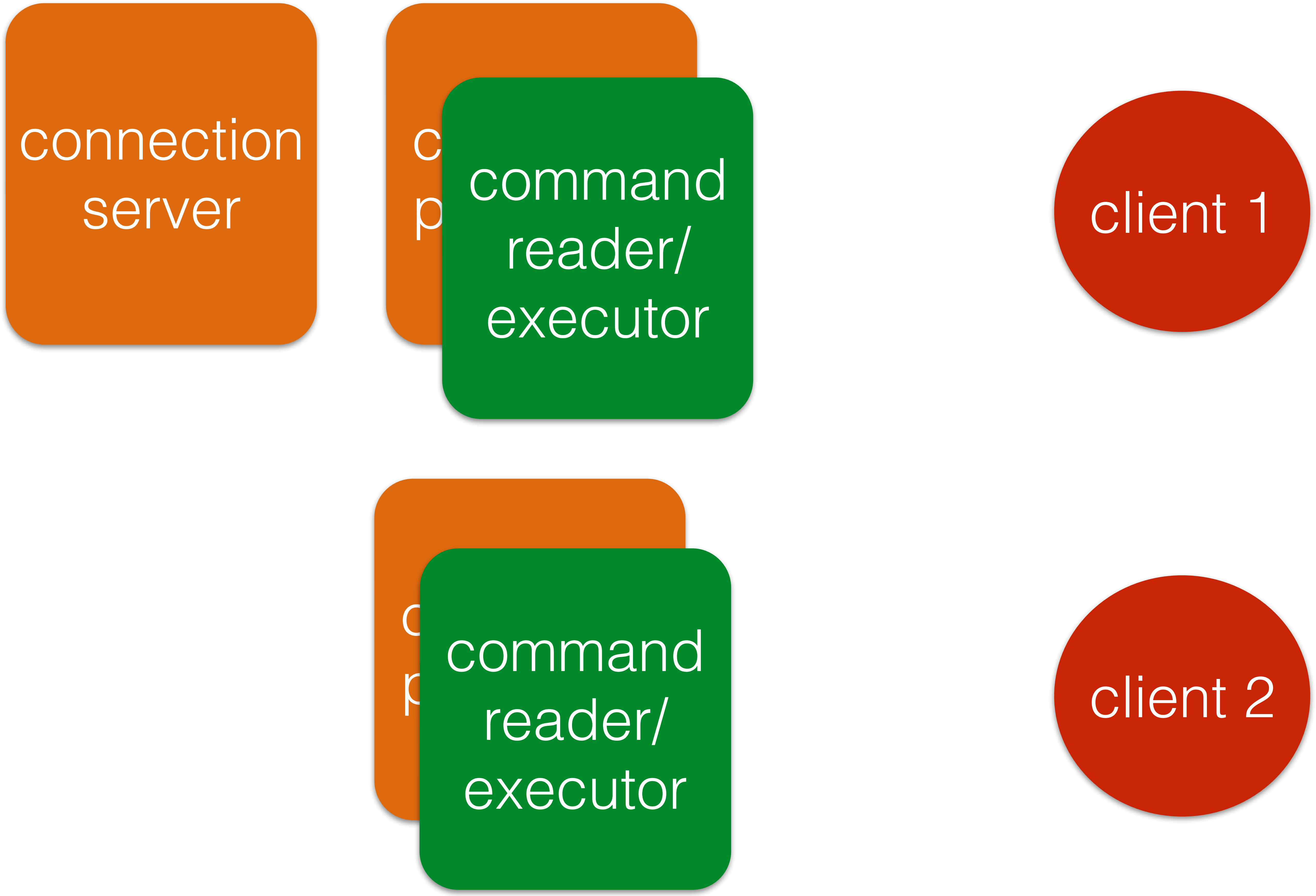
- **Command reader sandboxed**
  - Not root
  - Handles most commands
  - Except few requiring privilege
- **Comm. proc. only talks to reader**
  - And, again, white-lists its limited input

Attack: Cross-session

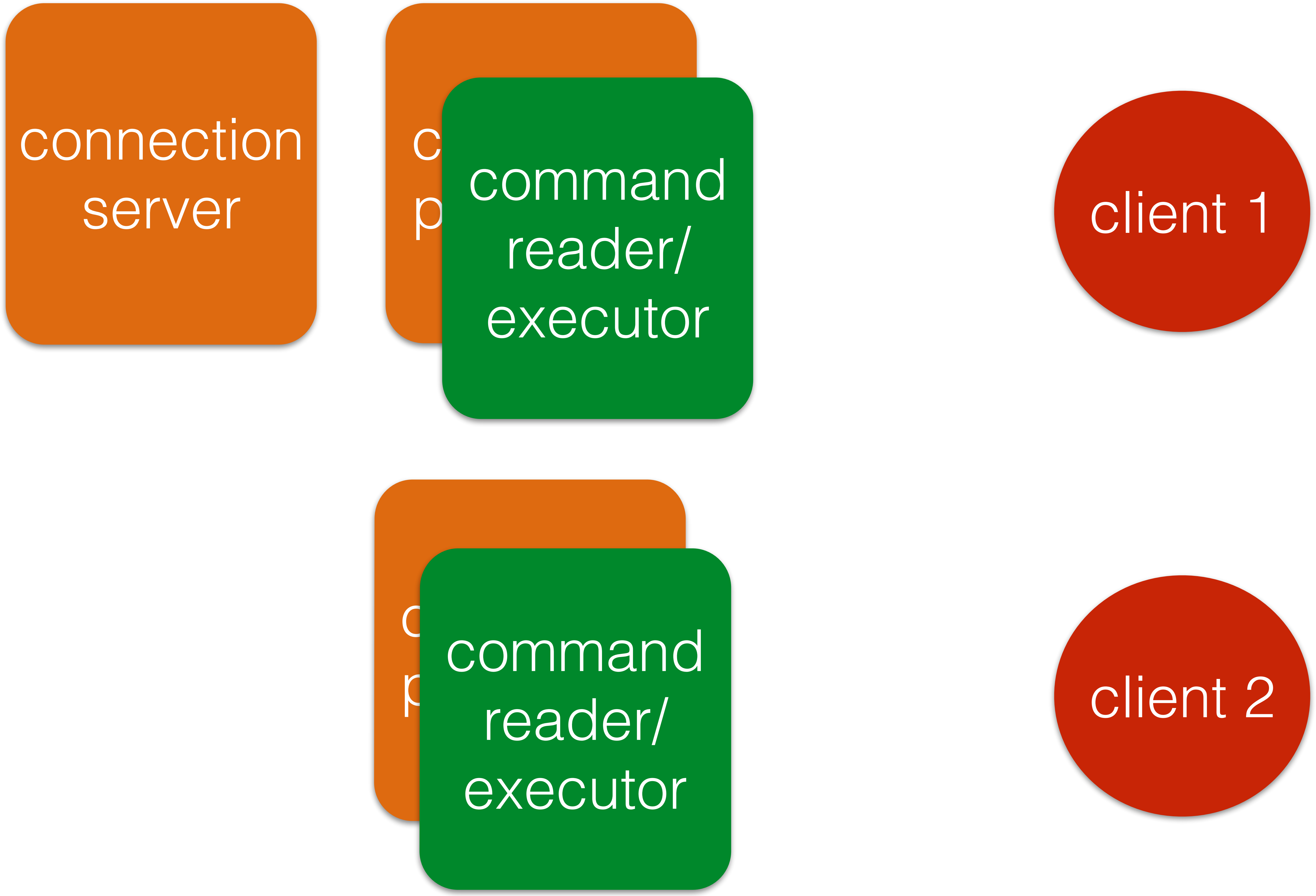
Attack: Cross-session



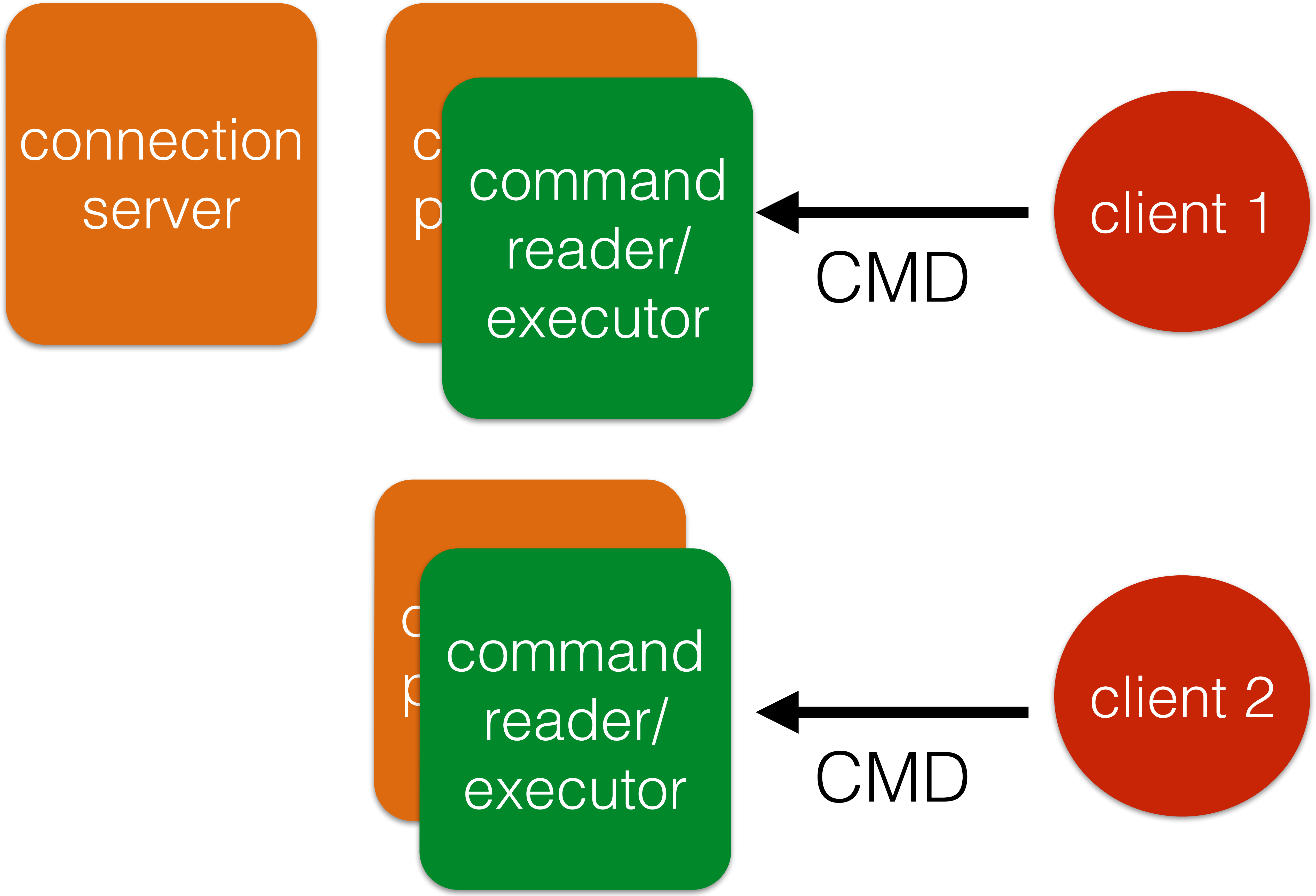
Attack: Cross-session



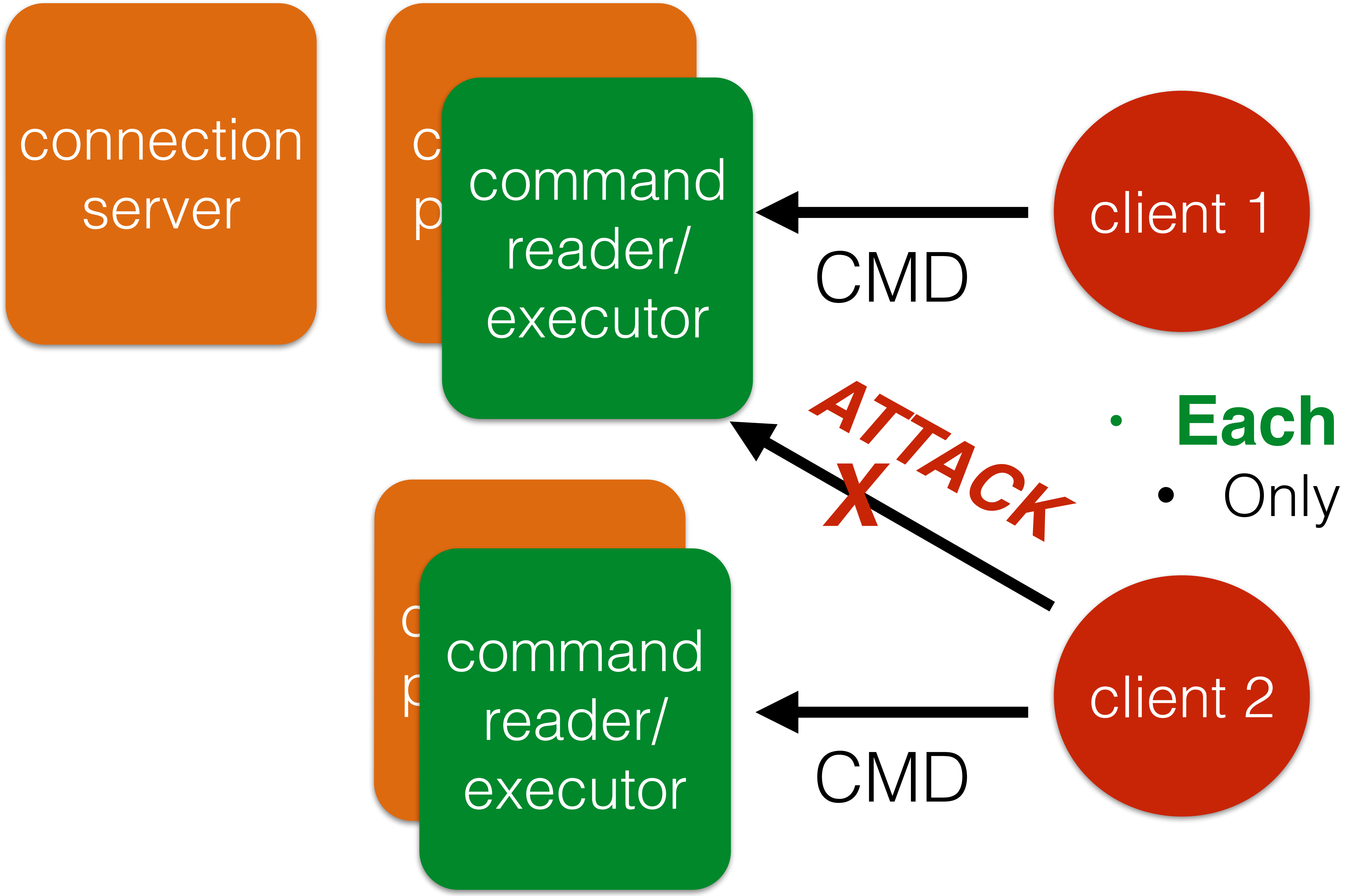
Attack: Cross-session



Attack: Cross-session



Attack: Cross-session



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

Presenting vsftpd's secure design

vsftpd employs a secure design. The UNIX facilities outlined above are used to good effect. The design decisions taken are as follows:

1) All parsing and acting on potentially malicious remote network data is done in a process running as an unprivileged user. Furthermore, this process runs in a chroot() jail, ensuring only the ftp files area is accessible.

2) Any privileged operations are handled in a privileged parent process. The code for this privileged parent process is as small as possible for safety.

3) This same privileged parent process receives requests from the unprivileged child over a socket. All requests are distrusted. Here are example requests:
- Login request. The child sends username and password. Only if the details are correct does the privileged parent launch a new child with the appropriate user credentials.
- chown() request. The child may request a recently uploaded file gets chown'ed() to root for security purposes. The parent is careful to only allow chown() to root, and only from files owned by the ftp user.
- Get privileged socket request. The ftp protocol says we are supposed to emit data connections from port 20. This requires privilege. The privileged parent process creates the privileged socket and passes it to child over the socket.

4) This same privileged parent process makes use of capabilities and chroot(), to run with the least privilege required. After login, depending on what options have been selected, the privileged parent dynamically calculates what privileges it requires. In some cases, this amounts to no privilege, and the privileged parent just exits, leaving no part of vsftpd running with privilege.

5) vsftpd-2.0.0 introduces SSL / TLS support using OpenSSL. ALL OpenSSL protocol parsing is performed in a chroot() jail, running under an unprivileged user. This means both pre-authenticated and post-authenticated OpenSSL protocol parsing; it's actually quite hard to do, but vsftpd manages it in the name of being secure. I'm unaware of any other FTP server which supports both SSL / TLS and privilege separatation, and gets this right.

Comments on this document are welcomed.

Presenting vsftpd's secure design

vsftpd employs a secure design. The UNIX facilities outlined above are used to good effect. The design decisions taken are as follows:

1) All parsing and acting on potentially malicious remote network data is done in a process running as an unprivileged user. Furthermore, this process runs in a chroot() jail, ensuring only the ftp files area is accessible.

2) Any privileged operations are handled in a privileged parent process. The code for this privileged parent process is as small as possible for safety.

3) This same privileged parent process receives requests from the unprivileged child over a socket. All requests are distrusted. Here are example requests:
- Login request. The child sends username and password. Only if the details are correct does the privileged parent launch a new child with the appropriate user credentials.
- chown() request. The child may request a recently uploaded file gets chown'ed() to root for security purposes. The parent is careful to only allow chown() to root, and only from files owned by the ftp user.
- Get privileged socket request. The ftp protocol says we are supposed to emit data connections from port 20. This requires privilege. The privileged parent process creates the privileged socket and passes it to child over the socket.

4) This same privileged parent process makes use of capabilities and chroot(), to run with the least privilege required. After login, depending on what options have been selected, the privileged parent dynamically calculates what privileges it requires. In some cases, this amounts to no privilege, and the privileged parent just exits, leaving no part of vsftpd running with privilege.

5) vsftpd-2.0.0 introduces SSL / TLS support using OpenSSL. ALL OpenSSL protocol parsing is performed in a chroot() jail, running under an unprivileged user. This means both pre-authenticated and post-authenticated OpenSSL protocol parsing; it's actually quite hard to do, but vsftpd manages it in the name of being secure. I'm unaware of any other FTP server which supports both SSL / TLS and privilege separation, and gets this right.

Comments on this document are welcomed.

Separation of responsibilities

```
Presenting vsftpd's secure design
=================================

vsftpd employs a secure design. The UNIX facilities outlined above are used
to good effect. The design decisions taken are as follows:

1) All parsing and acting on potentially malicious remote network data is
done in a process running as an unprivileged user. Furthermore, this process
runs in a chroot() jail, ensuring only the ftp files area is accessible.

2) Any privileged operations are handled in a privileged parent process. The
code for this privileged parent process is as small as possible for safety.

3) This same privileged parent process receives requests from the unprivileged
child over a socket. All requests are distrusted. Here are example requests:
- Login request. The child sends username and password. Only if the details
are correct does the privileged parent launch a new child with the appropriate
user credentials.
- chown() request. The child may request a recently uploaded file gets
chown'ed() to root for security purposes. The parent is careful to only allow
chown() to root, and only from files owned by the ftp user.
- Get privileged socket request. The ftp protocol says we are supposed to
emit data connections from port 20. This requires privilege. The privileged
parent process creates the privileged socket and passes it to child over
the socket.

4) This same privileged parent process makes use of capabilities and chroot(),
to run with the least privilege required. After login, depending on what
options have been selected, the privileged parent dynamically calculates what
privileges it requires. In some cases, this amounts to no privilege, and the
privileged parent just exits, leaving no part of vsftpd running with
privilege.

5) vsftpd-2.0.0 introduces SSL / TLS support using OpenSSL. ALL OpenSSL
protocol parsing is performed in a chroot() jail, running under an unprivileged
user. This means both pre-authenticated and post-authenticated OpenSSL protocol
parsing; it's actually quite hard to do, but vsftpd manages it in the name of
being secure. I'm unaware of any other FTP server which supports both SSL / TLS
and privilege separation, and gets this right.

Comments on this document are welcomed.
```

Separation of responsibilities

```
Presenting vsftpd's secure design
=================================

vsftpd employs a secure design. The UNIX facilities outlined above are used
to good effect. The design decisions taken are as follows:

1) All parsing and acting on potentially malicious remote network data is
done in a process running as an unprivileged user. Furthermore, this process
runs in a chroot() jail, ensuring only the ftp files area is accessible.

2) Any privileged operations are handled in a privileged parent process. The
code for this privileged parent process is as small as possible for safety.

3) This same privileged parent process receives requests from the unprivileged
child over a socket. All requests are distrusted. Here are example requests:
- Login request. The child sends username and password. Only if the details
are correct does the privileged parent launch a new child with the appropriate
user credentials.
- chown() request. The child may request a recently uploaded file gets
chown'ed() to root for security purposes. The parent is careful to only allow
chown() to root, and only from files owned by the ftp user.
- Get privileged socket request. The ftp protocol says we are supposed to
emit data connections from port 20. This requires privilege. The privileged
parent process creates the privileged socket and passes it to child over
the socket.

4) This same privileged parent process makes use of capabilities and chroot(),
to run with the least privilege required. After login, depending on what
options have been selected, the privileged parent dynamically calculates what
privileges it requires. In some cases, this amounts to no privilege, and the
privileged parent just exits, leaving no part of vsftpd running with
privilege.

5) vsftpd-2.0.0 introduces SSL / TLS support using OpenSSL. ALL OpenSSL
protocol parsing is performed in a chroot() jail, running under an unprivileged
user. This means both pre-authenticated and post-authenticated OpenSSL protocol
parsing; it's actually quite hard to do, but vsftpd manages it in the name of
being secure. I'm unaware of any other FTP server which supports both SSL / TLS
and privilege separatation, and gets this right.

Comments on this document are welcomed.
```

Separation of responsibilities

```
Presenting vsftpd's secure design
=================================

vsftpd employs a secure design. The UNIX facilities outlined above are used
to good effect. The design decisions taken are as follows:

1) All parsing and acting on potentially malicious remote network data is
done in a process running as an unprivileged user. Furthermore, this process
runs in a chroot() jail, ensuring only the ftp files area is accessible.

2) Any privileged operations are handled in a privileged parent process. The
code for this privileged parent process is as small as possible for safety.

3) This same privileged parent process receives requests from the unprivileged
child over a socket. All requests are distrusted. Here are example requests:
- Login request. The child sends username and password. Only if the details
are correct does the privileged parent launch a new child with the appropriate
user credentials.
- chown() request. The child may request a recently uploaded file gets
chown'ed() to root for security purposes. The parent is careful to only allow
chown() to root, and only from files owned by the ftp user.
- Get privileged socket request. The ftp protocol says we are supposed to
emit data connections from port 20. This requires privilege. The privileged
parent process creates the privileged socket and passes it to child over
the socket.

4) This same privileged parent process makes use of capabilities and chroot(),
to run with the least privilege required. After login, depending on what
options have been selected, the privileged parent dynamically calculates what
privileges it requires. In some cases, this amounts to no privilege, and the
privileged parent just exits, leaving no part of vsftpd running with
privilege.

5) vsftpd-2.0.0 introduces SSL / TLS support using OpenSSL. ALL OpenSSL
protocol parsing is performed in a chroot() jail, running under an unprivileged
user. This means both pre-authenticated and post-authenticated OpenSSL protocol
parsing; it's actually quite hard to do, but vsftpd manages it in the name of
being secure. I'm unaware of any other FTP server which supports both SSL / TLS
and privilege separation, and gets this right.

Comments on this document are welcomed.
```

Separation of responsibilities

TCB: KISS

```
Presenting vsftpd's secure design
=================================

vsftpd employs a secure design. The UNIX facilities outlined above are used
to good effect. The design decisions taken are as follows:

1) All parsing and acting on potentially malicious remote network data is
done in a process running as an unprivileged user. Furthermore, this process
runs in a chroot() jail, ensuring only the ftp files area is accessible.

2) Any privileged operations are handled in a privileged parent process. The
code for this privileged parent process is as small as possible for safety.

3) This same privileged parent process receives requests from the unprivileged
child over a socket. All requests are distrusted. Here are example requests:
- Login request. The child sends username and password. Only if the details
are correct does the privileged parent launch a new child with the appropriate
user credentials.
- chown() request. The child may request a recently uploaded file gets
chown'ed() to root for security purposes. The parent is careful to only allow
chown() to root, and only from files owned by the ftp user.
- Get privileged socket request. The ftp protocol says we are supposed to
emit data connections from port 20. This requires privilege. The privileged
parent process creates the privileged socket and passes it to child over
the socket.

4) This same privileged parent process makes use of capabilities and chroot(),
to run with the least privilege required. After login, depending on what
options have been selected, the privileged parent dynamically calculates what
privileges it requires. In some cases, this amounts to no privilege, and the
privileged parent just exits, leaving no part of vsftpd running with
privilege.

5) vsftpd-2.0.0 introduces SSL / TLS support using OpenSSL. ALL OpenSSL
protocol parsing is performed in a chroot() jail, running under an unprivileged
user. This means both pre-authenticated and post-authenticated OpenSSL protocol
parsing; it's actually quite hard to do, but vsftpd manages it in the name of
being secure. I'm unaware of any other FTP server which supports both SSL / TLS
and privilege separation, and gets this right.

Comments on this document are welcomed.
```

Separation of responsibilities

TCB: KISS

```
Presenting vsftpd's secure design
=================================

vsftpd employs a secure design. The UNIX facilities outlined above are used
to good effect. The design decisions taken are as follows:

1) All parsing and acting on potentially malicious remote network data is
done in a process running as an unprivileged user. Furthermore, this process
runs in a chroot() jail, ensuring only the ftp files area is accessible.

2) Any privileged operations are handled in a privileged parent process. The
code for this privileged parent process is as small as possible for safety.

3) This same privileged parent process receives requests from the unprivileged
child over a socket. All requests are distrusted. Here are example requests:
- Login request. The child sends username and password. Only if the details
are correct does the privileged parent launch a new child with the appropriate
user credentials.
- chown() request. The child may request a recently uploaded file gets
chown'ed() to root for security purposes. The parent is careful to only allow
chown() to root, and only from files owned by the ftp user.
- Get privileged socket request. The ftp protocol says we are supposed to
emit data connections from port 20. This requires privilege. The privileged
parent process creates the privileged socket and passes it to child over
the socket.

4) This same privileged parent process makes use of capabilities and chroot(),
to run with the least privilege required. After login, depending on what
options have been selected, the privileged parent dynamically calculates what
privileges it requires. In some cases, this amounts to no privilege, and the
privileged parent just exits, leaving no part of vsftpd running with
privilege.

5) vsftpd-2.0.0 introduces SSL / TLS support using OpenSSL. ALL OpenSSL
protocol parsing is performed in a chroot() jail, running under an unprivileged
user. This means both pre-authenticated and post-authenticated OpenSSL protocol
parsing; it's actually quite hard to do, but vsftpd manages it in the name of
being secure. I'm unaware of any other FTP server which supports both SSL / TLS
and privilege separatation, and gets this right.

Comments on this document are welcomed.
```

Separation of responsibilities

TCB: KISS

Presenting vsftpd's secure design

vsftpd employs a secure design. The UNIX facilities outlined above are used to good effect. The design decisions taken are as follows:

1) All parsing and acting on potentially malicious remote network data is done in a process running as an unprivileged user. Furthermore, this process runs in a chroot() jail, ensuring only the ftp files area is accessible.

2) Any privileged operations are handled in a privileged parent process. The code for this privileged parent process is as small as possible for safety.

3) This same privileged parent process receives requests from the unprivileged child over a socket. All requests are distrusted. Here are example requests:
- Login request. The child sends username and password. Only if the details are correct does the privileged parent launch a new child with the appropriate user credentials.
- chown() request. The child may request a recently uploaded file gets chown'ed() to root for security purposes. The parent is careful to only allow chown() to root, and only from files owned by the ftp user.
- Get privileged socket request. The ftp protocol says we are supposed to emit data connections from port 20. This requires privilege. The privileged parent process creates the privileged socket and passes it to child over the socket.

4) This same privileged parent process makes use of capabilities and chroot(), to run with the least privilege required. After login, depending on what options have been selected, the privileged parent dynamically calculates what privileges it requires. In some cases, this amounts to no privilege, and the privileged parent just exits, leaving no part of vsftpd running with privilege.

5) vsftpd-2.0.0 introduces SSL / TLS support using OpenSSL. ALL OpenSSL protocol parsing is performed in a chroot() jail, running under an unprivileged user. This means both pre-authenticated and post-authenticated OpenSSL protocol parsing; it's actually quite hard to do, but vsftpd manages it in the name of being secure. I'm unaware of any other FTP server which supports both SSL / TLS and privilege separation, and gets this right.

Comments on this document are welcomed.

Separation of responsibilities

TCB: KISS

TCB: Privilege separation

```
Presenting vsftpd's secure design
=================================

vsftpd employs a secure design. The UNIX facilities outlined above are used
to good effect. The design decisions taken are as follows:

1) All parsing and acting on potentially malicious remote network data is
done in a process running as an unprivileged user. Furthermore, this process
runs in a chroot() jail, ensuring only the ftp files area is accessible.

2) Any privileged operations are handled in a privileged parent process. The
code for this privileged parent process is as small as possible for safety.

3) This same privileged parent process receives requests from the unprivileged
child over a socket. All requests are distrusted. Here are example requests:
- Login request. The child sends username and password. Only if the details
are correct does the privileged parent launch a new child with the appropriate
user credentials.
- chown() request. The child may request a recently uploaded file gets
chown'ed() to root for security purposes. The parent is careful to only allow
chown() to root, and only from files owned by the ftp user.
- Get privileged socket request. The ftp protocol says we are supposed to
emit data connections from port 20. This requires privilege. The privileged
parent process creates the privileged socket and passes it to child over
the socket.

4) This same privileged parent process makes use of capabilities and chroot(),
to run with the least privilege required. After login, depending on what
options have been selected, the privileged parent dynamically calculates what
privileges it requires. In some cases, this amounts to no privilege, and the
privileged parent just exits, leaving no part of vsftpd running with
privilege.

5) vsftpd-2.0.0 introduces SSL / TLS support using OpenSSL. ALL OpenSSL
protocol parsing is performed in a chroot() jail, running under an unprivileged
user. This means both pre-authenticated and post-authenticated OpenSSL protocol
parsing; it's actually quite hard to do, but vsftpd manages it in the name of
being secure. I'm unaware of any other FTP server which supports both SSL / TLS
and privilege separation, and gets this right.

Comments on this document are welcomed.
```

Separation of responsibilities

TCB: KISS

TCB: Privilege separation

```
Presenting vsftpd's secure design
=================================

vsftpd employs a secure design. The UNIX facilities outlined above are used
to good effect. The design decisions taken are as follows:

1) All parsing and acting on potentially malicious remote network data is
done in a process running as an unprivileged user. Furthermore, this process
runs in a chroot() jail, ensuring only the ftp files area is accessible.

2) Any privileged operations are handled in a privileged parent process. The
code for this privileged parent process is as small as possible for safety.

3) This same privileged parent process receives requests from the unprivileged
child over a socket. All requests are distrusted. Here are example requests:
- Login request. The child sends username and password. Only if the details
are correct does the privileged parent launch a new child with the appropriate
user credentials.
- chown() request. The child may request a recently uploaded file gets
chown'ed() to root for security purposes. The parent is careful to only allow
chown() to root, and only from files owned by the ftp user.
- Get privileged socket request. The ftp protocol says we are supposed to
emit data connections from port 20. This requires privilege. The privileged
parent process creates the privileged socket and passes it to child over
the socket.

4) This same privileged parent process makes use of capabilities and chroot(),
to run with the least privilege required. After login, depending on what
options have been selected, the privileged parent dynamically calculates what
privileges it requires. In some cases, this amounts to no privilege, and the
privileged parent just exits, leaving no part of vsftpd running with
privilege.

5) vsftpd-2.0.0 introduces SSL / TLS support using OpenSSL. ALL OpenSSL
protocol parsing is performed in a chroot() jail, running under an unprivileged
user. This means both pre-authenticated and post-authenticated OpenSSL protocol
parsing; it's actually quite hard to do, but vsftpd manages it in the name of
being secure. I'm unaware of any other FTP server which supports both SSL / TLS
and privilege separatation, and gets this right.

Comments on this document are welcomed.
```

Separation of responsibilities

TCB: KISS

TCB: Privilege separation

Presenting vsftpd's secure design
====================================

vsftpd employs a secure design. The UNIX facilities outlined above are used to good effect. The design decisions taken are as follows:

1) All parsing and acting on potentially malicious remote network data is done in a process running as an unprivileged user. Furthermore, this process runs in a chroot() jail, ensuring only the ftp files area is accessible.

2) Any privileged operations are handled in a privileged parent process. The code for this privileged parent process is as small as possible for safety.

3) This same privileged parent process receives requests from the unprivileged child over a socket. All requests are distrusted. Here are example requests:
- Login request. The child sends username and password. Only if the details are correct does the privileged parent launch a new child with the appropriate user credentials.
- chown() request. The child may request a recently uploaded file gets chown'ed() to root for security purposes. The parent is careful to only allow chown() to root, and only from files owned by the ftp user.
- Get privileged socket request. The ftp protocol says we are supposed to emit data connections from port 20. This requires privilege. The privileged parent process creates the privileged socket and passes it to child over the socket.

4) This same privileged parent process makes use of capabilities and chroot(), to run with the least privilege required. After login, depending on what options have been selected, the privileged parent dynamically calculates what privileges it requires. In some cases, this amounts to no privilege, and the privileged parent just exits, leaving no part of vsftpd running with privilege.

5) vsftpd-2.0.0 introduces SSL / TLS support using OpenSSL. ALL OpenSSL protocol parsing is performed in a chroot() jail, running under an unprivileged user. This means both pre-authenticated and post-authenticated OpenSSL protocol parsing; it's actually quite hard to do, but vsftpd manages it in the name of being secure. I'm unaware of any other FTP server which supports both SSL / TLS and privilege separation, and gets this right.

Comments on this document are welcomed.

Separation of responsibilities

TCB: KISS

TCB: Privilege separation

Principle of least privilege

```
Presenting vsftpd's secure design
=================================

vsftpd employs a secure design. The UNIX facilities outlined above are used
to good effect. The design decisions taken are as follows:

1) All parsing and acting on potentially malicious remote network data is
done in a process running as an unprivileged user. Furthermore, this process
runs in a chroot() jail, ensuring only the ftp files area is accessible.

2) Any privileged operations are handled in a privileged parent process. The
code for this privileged parent process is as small as possible for safety.

3) This same privileged parent process receives requests from the unprivileged
child over a socket. All requests are distrusted. Here are example requests:
- Login request. The child sends username and password. Only if the details
are correct does the privileged parent launch a new child with the appropriate
user credentials.
- chown() request. The child may request a recently uploaded file gets
chown'ed() to root for security purposes. The parent is careful to only allow
chown() to root, and only from files owned by the ftp user.
- Get privileged socket request. The ftp protocol says we are supposed to
emit data connections from port 20. This requires privilege. The privileged
parent process creates the privileged socket and passes it to child over
the socket.

4) This same privileged parent process makes use of capabilities and chroot(),
to run with the least privilege required. After login, depending on what
options have been selected, the privileged parent dynamically calculates what
privileges it requires. In some cases, this amounts to no privilege, and the
privileged parent just exits, leaving no part of vsftpd running with
privilege.

5) vsftpd-2.0.0 introduces SSL / TLS support using OpenSSL. ALL OpenSSL
protocol parsing is performed in a chroot() jail, running under an unprivileged
user. This means both pre-authenticated and post-authenticated OpenSSL protocol
parsing; it's actually quite hard to do, but vsftpd manages it in the name of
being secure. I'm unaware of any other FTP server which supports both SSL / TLS
and privilege separation, and gets this right.

Comments on this document are welcomed.
```

Presenting vsftpd's secure design

vsftpd employs a secure design. The UNIX facilities outlined above are used to good effect. The design decisions taken are as follows:

Separation of responsibilities

1) All parsing and acting on potentially malicious remote network data is done in a process running as an unprivileged user. Furthermore, this process runs in a chroot() jail, ensuring only the ftp files area is accessible.

TCB: KISS

2) Any privileged operations are handled in a privileged parent process. The code for this privileged parent process is as small as possible for safety.

TCB: Privilege separation

3) This same privileged parent process receives requests from the unprivileged child over a socket. All requests are distrusted. Here are example requests:
- Login request. The child sends username and password. Only if the details are correct does the privileged parent launch a new child with the appropriate user credentials.
- chown() request. The child may request a recently uploaded file gets chown'ed() to root for security purposes. The parent is careful to only allow chown() to root, and only from files owned by the ftp user.
- Get privileged socket request. The ftp protocol says we are supposed to emit data connections from port 20. This requires privilege. The privileged parent process creates the privileged socket and passes it to child over the socket.

Principle of least privilege

4) This same privileged parent process makes use of capabilities and chroot(), to run with the least privilege required. After login, depending on what options have been selected, the privileged parent dynamically calculates what privileges it requires. In some cases, this amounts to no privilege, and the privileged parent just exits, leaving no part of vsftpd running with privilege.

5) vsftpd-2.0.0 introduces SSL / TLS support using OpenSSL. ALL OpenSSL protocol parsing is performed in a chroot() jail, running under an unprivileged user. This means both pre-authenticated and post-authenticated OpenSSL protocol parsing; it's actually quite hard to do, but vsftpd manages it in the name of being secure. I'm unaware of any other FTP server which supports both SSL / TLS and privilege separatation, and gets this right.

Comments on this document are welcomed.

```
Presenting vsftpd's secure design
=================================

vsftpd employs a secure design. The UNIX facilities outlined above are used
to good effect. The design decisions taken are as follows:

1) All parsing and acting on potentially malicious remote network data is
done in a process running as an unprivileged user. Furthermore, this process
runs in a chroot() jail, ensuring only the ftp files area is accessible.

2) Any privileged operations are handled in a privileged parent process. The
code for this privileged parent process is as small as possible for safety.

3) This same privileged parent process receives requests from the unprivileged
child over a socket. All requests are distrusted. Here are example requests:
- Login request. The child sends username and password. Only if the details
are correct does the privileged parent launch a new child with the appropriate
user credentials.
- chown() request. The child may request a recently uploaded file gets
chown'ed() to root for security purposes. The parent is careful to only allow
chown() to root, and only from files owned by the ftp user.
- Get privileged socket request. The ftp protocol says we are supposed to
emit data connections from port 20. This requires privilege. The privileged
parent process creates the privileged socket and passes it to child over
the socket.

4) This same privileged parent process makes use of capabilities and chroot(),
to run with the least privilege required. After login, depending on what
options have been selected, the privileged parent dynamically calculates what
privileges it requires. In some cases, this amounts to no privilege, and the
privileged parent just exits, leaving no part of vsftpd running with
privilege.

5) vsftpd-2.0.0 introduces SSL / TLS support using OpenSSL. ALL OpenSSL
protocol parsing is performed in a chroot() jail, running under an unprivileged
user. This means both pre-authenticated and post-authenticated OpenSSL protocol
parsing; it's actually quite hard to do, but vsftpd manages it in the name of
being secure. I'm unaware of any other FTP server which supports both SSL / TLS
and privilege separatation, and gets this right.

Comments on this document are welcomed.
```

Separation of responsibilities

TCB: KISS

TCB: Privilege separation

Principle of least privilege

```
Presenting vsftpd's secure design
=================================

vsftpd employs a secure design. The UNIX facilities outlined above are used
to good effect. The design decisions taken are as follows:

1) All parsing and acting on potentially malicious remote network data is
done in a process running as an unprivileged user. Furthermore, this process
runs in a chroot() jail, ensuring only the ftp files area is accessible.

2) Any privileged operations are handled in a privileged parent process. The
code for this privileged parent process is as small as possible for safety.

3) This same privileged parent process receives requests from the unprivileged
child over a socket. All requests are distrusted. Here are example requests:
- Login request. The child sends username and password. Only if the details
are correct does the privileged parent launch a new child with the appropriate
user credentials.
- chown() request. The child may request a recently uploaded file gets
chown'ed() to root for security purposes. The parent is careful to only allow
chown() to root, and only from files owned by the ftp user.
- Get privileged socket request. The ftp protocol says we are supposed to
emit data connections from port 20. This requires privilege. The privileged
parent process creates the privileged socket and passes it to child over
the socket.

4) This same privileged parent process makes use of capabilities and chroot(),
to run with the least privilege required. After login, depending on what
options have been selected, the privileged parent dynamically calculates what
privileges it requires. In some cases, this amounts to no privilege, and the
privileged parent just exits, leaving no part of vsftpd running with
privilege.

5) vsftpd-2.0.0 introduces SSL / TLS support using OpenSSL. ALL OpenSSL
protocol parsing is performed in a chroot() jail, running under an unprivileged
user. This means both pre-authenticated and post-authenticated OpenSSL protocol
parsing; it's actually quite hard to do, but vsftpd manages it in the name of
being secure. I'm unaware of any other FTP server which supports both SSL / TLS
and privilege separatation, and gets this right.

Comments on this document are welcomed.
```

Separation of responsibilities

TCB: KISS

TCB: Privilege separation

Principle of least privilege

Kerkhoff's principle!

Reasoning about code safety

Reasoning about code safety

Reasoning about code safety

- **Goal**: Confidence that our code is safe and correct

Reasoning about code safety

- **Goal**: Confidence that our code is safe and correct

Reasoning about code safety

- **Goal**: Confidence that our code is safe and correct
- **Approach**: Build up this confidence function by function, module by module

Reasoning about code safety

- **Goal**: Confidence that our code is safe and correct
- **Approach**: Build up this confidence function by function, module by module

Reasoning about code safety

- **Goal**: Confidence that our code is safe and correct
- **Approach**: Build up this confidence function by function, module by module
- **Modularity provides boundaries for our reasoning**

Reasoning about code safety

- **Goal**: Confidence that our code is safe and correct
- **Approach**: Build up this confidence function by function, module by module
- **Modularity provides boundaries for our reasoning**
  - **Preconditions**: what must hold to be correct (“**REQUIRES**”)
  - **Postconditions**: what holds after the function (“**ENSURES**”)

Reasoning about code safety

- **Goal**: Confidence that our code is safe and correct
- **Approach**: Build up this confidence function by function, module by module
- **Modularity provides boundaries for our reasoning**
  - <u>**Preconditions**</u>: what must hold to be correct (“**REQUIRES**”)
  - <u>**Postconditions**</u>: what holds after the function (“**ENSURES**”)

Reasoning about code safety

- **Goal**: Confidence that our code is safe and correct
- **Approach**: Build up this confidence function by function, module by module
- **Modularity provides boundaries for our reasoning**
  - **Preconditions**: what must hold to be correct ("**REQUIRES**")
  - **Postconditions**: what holds after the function ("**ENSURES**")
- Think of it as a **contract** for using the module
  - "Statement 1's postcondition should meet statement 2's precondition"

Reasoning about code safety

- **Goal**: Confidence that our code is safe and correct
- **Approach**: Build up this confidence function by function, module by module
- **Modularity provides boundaries for our reasoning**
  - **Preconditions**: what must hold to be correct ("**REQUIRES**")
  - **Postconditions**: what holds after the function ("**ENSURES**")
- Think of it as a **contract** for using the module
  - "Statement 1's postcondition should meet statement 2's precondition"
- **Invariant = Conditions that always hold within some part of a function**

What are the preconditions to ensure safety?

```c
/* requires:                                            */
/* ensures: retval is the first four bytes p pointed to */

int deref(int *p) {
    return *p;
}
```

What are the preconditions to ensure safety?

```
/* requires: p != NULL (and p is a valid pointer)       */
/* ensures: retval is the first four bytes p pointed to */

int deref(int *p) {
    return *p;
}
```

What are the postconditions to ensure safety?

```
/* ensures:                                          */

void *myalloc(size_t *n) {
    void *p = malloc(n);
    if (!p) { perror("malloc"); exit(1); }
    return p;
}
```

What are the postconditions to ensure safety?

```
/* ensures: retval != NULL (and a valid pointer)   */

void *myalloc(size_t *n) {
    void *p = malloc(n);
    if (!p) { perror(“malloc”); exit(1); }
    return p;
}
```

What are the preconditions to ensure safety?

```
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += a[i];
    return total;
}
```

What are the preconditions to ensure safety?

Approach:

1. Identify each memory access
2. Annotate with preconditions it requires
3. Propagate the requirements up

```
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += a[i];
    return total;
}
```

What are the preconditions to ensure safety?

Approach:

1. Identify each memory access
2. Annotate with preconditions it requires
3. Propagate the requirements up

```
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += a[i];
    return total;
}
```

Memory access

What are the preconditions to ensure safety?

Approach:
1. Identify each memory access
2. Annotate with preconditions it requires
3. Propagate the requirements up

```
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += a[i];
    return total;
}
```

Memory access

```
/* requires: a != NULL   */
/* requires: 0 <= i      */
/* requires: i < size(a) */
```

What are the preconditions to ensure safety?

Approach:

1. Identify each memory access
2. Annotate with preconditions it requires
3. Propagate the requirements up

```c
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += a[i];
    return total;
}
```

Memory access

```c
/* requires: a != NULL    */
/* requires: 0 <= i       */
/* requires: i < size(a) */
```

What are the preconditions to ensure safety?

Approach:

1. Identify each memory access
2. Annotate with preconditions it requires
3. Propagate the requirements up

```
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += a[i];
    return total;
}
```

Memory access

No line of code above this guarantees it will hold: so move it up

```
/* requires: a != NULL    */
/* requires: 0 <= i       */
/* requires: i < size(a) */
```

What are the preconditions to ensure safety?

Approach:

1. Identify each memory access
2. Annotate with preconditions it requires
3. Propagate the requirements up

```
/* requires: a != NULL    */


int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += a[i];
    return total;
}
```

Memory access

```
/* requires: 0 <= i       */
/* requires: i < size(a) */
```

What are the preconditions to ensure safety?

Approach:
1. Identify each memory access
2. Annotate with preconditions it requires
3. Propagate the requirements up

```
/* requires: a != NULL   */


int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += a[i];
    return total;
}
```

Memory access

```
/* requires: 0 <= i      */
/* requires: i < size(a) */
```

What are the preconditions to ensure safety?

Approach:
1. Identify each memory access
2. Annotate with preconditions it requires
3. Propagate the requirements up

```
/* requires: a != NULL    */

int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += a[i];
    return total;
}
```

Memory access

Line above it: `size_t i` ensures that `0 <= i` always

```
/* requires: 0 <= i      */
/* requires: i < size(a) */
```

What are the preconditions to ensure safety?

Approach:

1. Identify each memory access
2. Annotate with preconditions it requires
3. Propagate the requirements up

```
/* requires: a != NULL    */


int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += a[i];
    return total;
}
```

Memory access

Line above it: `size_t i` ensures that `0 <= i` always

```
/* requires: i < size(a) */
```

What are the preconditions to ensure safety?

Approach:

1. Identify each memory access
2. Annotate with preconditions it requires
3. Propagate the requirements up

```
/* requires: a != NULL    */


int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += a[i];
    return total;
}
```

Memory access

```
/* requires: i < size(a) */
```

What are the preconditions to ensure safety?

Approach:
1. Identify each memory access
2. Annotate with preconditions it requires
3. Propagate the requirements up

```c
/* requires: a != NULL    */


int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += a[i];
    return total;
}
```

Memory access

Not guaranteed by above code

```c
/* requires: i < size(a) */
```

What are the preconditions to ensure safety?

Approach:

1. Identify each memory access
2. Annotate with preconditions it requires
3. Propagate the requirements up

```
/* requires: a != NULL    */


int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += a[i];
    return total;
}
```

```
/* requires: n < size(a) */
```

Memory access

Not guaranteed by above code

```
/* requires: i < size(a) */
```

What are the preconditions to ensure safety?

Approach:

1. Identify each memory access
2. Annotate with preconditions it requires
3. Propagate the requirements up

```
/* requires: a != NULL    */

/* requires: n < size(a) */

int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += a[i];
    return total;
}
```

Memory access

```c
char *tbl[N];  /* N is of type int */

/* requires:                                                    */
/* ensures:                                                     */
int hash(char *s) {
    int h = 17;
    while (*s)
        h = 257*h + (*s++) + 3;
    return h % N;
}


/* requires:                                                    */
bool search(char *s) {
    int i = hash(s);
    return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

```c
char *tbl[N];  /* N is of type int */

/* requires: s != NULL and valid, and NULL-terminated         */
/* ensures:                                                     */
int hash(char *s) {
    int h = 17;
    while (*s)
        h = 257*h + (*s++) + 3;
    return h % N;
}


/* requires:                                                     */
bool search(char *s) {
    int i = hash(s);
    return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

```c
char *tbl[N];  /* N is of type int */

/* requires: s != NULL and valid, and NULL-terminated          */
/* ensures:  0 <= retval < N                                   */
int hash(char *s) {
    int h = 17;
    while (*s)
        h = 257*h + (*s++) + 3;
    return h % N;
}


/* requires:                                                     */
bool search(char *s) {
    int i = hash(s);
    return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

```c
char *tbl[N];  /* N is of type int */

/* requires: s != NULL and valid, and NULL-terminated         */
/* ensures:  0 <= retval < N                                  */
int hash(char *s) {
    int h = 17;
    while (*s)
        h = 257*h + (*s++) + 3;
    return h % N;
}


/* requires: s != NULL (and a valid) and 0 <= hash < size(tbl) */
bool search(char *s) {
    int i = hash(s);
    return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

```c
char *tbl[N];  /* N is of type int */

/* requires: s != NULL and valid, and NULL-terminated         */
/* ensures:  0 <= retval < N                                   */
int hash(char *s) {
    int h = 17;
    while (*s)
        h = 257*h + (*s++) + 3;
    return h % N;
}


/* requires: s != NULL (and a valid) and 0 <= hash < size(tbl) */
bool search(char *s) {
    int i = hash(s);
    return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

Does this code meet its postconditions?

```c
char *tbl[N];  /* N is of type int */

/* requires: s != NULL and valid, and NULL-terminated         */
/* ensures:  0 <= retval < N                                  */
int hash(char *s) {
    int h = 17;
    while (*s)
        h = 257*h + (*s++) + 3;
    return h % N;
}


/* requires: s != NULL (and a valid) and 0 <= hash < size(tbl) */
bool search(char *s) {
    int i = hash(s);
    return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

Does this code meet its postconditions?   Need to change `int` to `unsigned int`

Why use pre & postconditions?

- Serves as documentation
- It allows **modular reasoning:** you can verify f() by only looking at
  - The code of f()
  - The annotations on every function that f() calls
- Thus, reasoning about a function's safety becomes an (almost) *purely local activity*
- This is **related to defensive programming**:
  - <u>Ideally</u>: preconditions are the assumptions we make
  - <u>Practically</u>: they're constraints that **honest** clients are expected to follow