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

October 2010 vulnerability watchlist

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

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

6 of the vulnerabilities are in security software


Approved for Public Release, Distribution Unlimited
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 \texttt{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-FTP, 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**
struct mystr
{
    char* PRIVATE_HANDS_OFF_p_buf;
    unsigned int PRIVATE_HANDS_OFF_len;
    unsigned int PRIVATE_HANDS_OFF_alloc_bytes;
};
Defense: Secure Strings

```c
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
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

```c
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

```c
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)
{
    ...
}

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
```
```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_memcmpy(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`
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;
}

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

Copy in at most \textit{len} bytes from \texttt{p\_src} into \texttt{p\_str}

\textbf{Consider NUL terminator when computing space.}
```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;
}
```

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

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

Copy in at most len bytes from p_src into p_str

consider NUL terminator when computing space

allocate space, if needed

copy in p_src contents
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 dereference 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;
}
```
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;
}
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
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
Connection Establishment

connection
server

client
Connection Establishment

TCP conn request
Connection Establishment

connection
server

command
processor

client
Connection Establishment

connection
server

command
processor

login
reader

client
Connection Establishment

- **connection server**
- **command processor**
- **login reader**
- **client**

U+P → OK

OK ← USER, PASS

OK
Connection Establishment

- connection server
- command processor
- command reader/executor
- client
Performing Commands

- connection server
- command processor
- command reader/executor
- client
Performing Commands

connection
server

command
processor

command
reader/
executor

client

OK

CHDIR
Performing Commands

connection server

command processor

command reader/executor

client

CHOWN

OK

OK

CHOWN
Logging out

connection server

command processor

command reader/executor

client
Logging out
Attack: Login

- connection server
- command processor
- login reader
- client
Attack: Login

connection server

command processor

login reader

client

ATTACK
**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

- connection server
- command processor
- command reader/executor
- client
Attack: Commands

connection server

command processor

command reader/executor

client

ATTACK
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

connection
server

client 1

client 2
Attack: Cross-session

connection server
command processor
command reader/executor

client 1
client 2
Attack: Cross-session

- connection server
- command reader/executor
- command processor
- client 1
- client 2
Attack: Cross-session

connection server

command reader/executor

client 1

command processor

client 2

command processor

command reader/executor
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 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:

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 omit 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

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'd() 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 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:

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.
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:

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:

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

Kerkhoff’s principle!

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.
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**

    • **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**
  - **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: 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 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;
}

/* 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;
}

int deref(int *p) {
    return *p;
}
What are the postconditions to ensure safety?

```c
/* 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 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?

```c
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

```c
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

```c
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

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

/* 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;
}
```

No line of code above this guarantees it will hold:

- `/* requires: a != NULL */` so move it up
- `/* 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;
}
```

/* 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;
}
```

/* requires: a != NULL */

```c
/* 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
/* 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;
}
```

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

```c
/* 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;
}
```

/* requires: a != NULL */

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

```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;
}
/* 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;
}

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

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

/* requires: a != NULL */

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

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;
}
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);
}
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);
}
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);
}
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);
}
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?
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**:
  • **Ideally**: preconditions are the assumptions we make
  • **Practically**: they’re constraints that **honest** clients are expected to follow