CyberSoft Operating Corporation Simple Virus Scanning Protocol (SVSP)

This document is based upon the Manual pages delivered with the VSTK product along with new sample code. Please note that some aspects of this protocol may not exist in older versions of the program. Enhancements to the protocol are made on a regular basis. It should be noted that VFindD supports multiple protocols in addition to SVSP.

NAME

SVSP - Simple Virus Scanning Protocol

DESCRIPTION

SVSP allows a client program to send data files to a VFindD server program for virus scanning. The protocol is a simple text based query/response, with asynchronous responses allowing simultaneous queueing of multiple files.

NOTES

Commands may be prefixed by a numerical identifier, if so, the responses to that command will be prefixed by the same identifier.

Filenames in command should not contain any quoting; file names in responses are quoted with double quotes; internal double quotes and backslashes are escaped with a backslash, newlines as \n.

After a SCAN command, SCANNING and possibly INFECTED responses are sent for each part of an archive/directory; for archives, the filename is then indicated as "archive" -> "part".

A QUEUED response may be sent if the scanning is delayed.

A DONE response is sent when all the parts of the scan request are complete. A client should not send additional SCAN commands until it has received a QUEUED, DONE, or Exxxxx response for the request.

A RECURSE limit of 0 implies unlimited (this is the default, if enabled).

For the IGNORE option, the virusname is as the virus name appears in the INFECTED response, e.g., CVDL W32/Nimda.E

If the EXPAND option is used without the expander value, uad is initiated with the default set of active expanders. The list of expanders and their default state are: tar (on), Z (on), gz (on), text (on), HTML

(off), zip (on), TNEF (on), Hqx8 (on), RAR (on), CAB (on). (List is increased on a regular basis pleaser refer to program notes for details or call CyberSoft technical support.

COMMANDS

[id] ENABLE option	Set a flag
[id] ENABLE option [value]	Add an option
[id] DISABLE option [value]	Clear a flag, remove options
[id] SCAN/how/what [args]	Scan data for viruses
[id] QUIT	Close and disconnect

OPTIONS

Options control details of the scanning process.

IGNORE virusname	Don't report this virus
EXCLUDE [filename]	Don't scan this file
EXPAND [expander]	Use this expander in UAD (* means all)
RECURSE limit	Expand files and directories recursively to the limit
ORIGINAL	Scan original or compressed file and components
HTML [1-3]	HTML to text conversion level

DATA SOURCE

The /what tells the source of data to scan.

/REQUEST [filename]	Client connects to server port
/PORT portnum [filename]	Server connects to client port
/FILE filename	Scan data from file/directory
/FILE-SHA1 filename	Scan data from file (& need its SHA1 hash)
/DATA bytes	Data sent inline, after SEND response **FUTURE**
(none)	Inline as above, until socket closing

Note: Only /FILE is implemented by the first version of the vfindd daemon.

DATA FORMAT

The /how indicates the format of the data to scan.

(none)	Raw data to be scanned

RESPONSES

The client should accept these responses:

[id] READY	Initial connection prompt
[id] DENIED	The connection is denied/blocked
[id] OK	Command accepted (option enabled, etc)
[id] QUEUED	File queued for later scanning
[id] Exxxxx message	Command failed (errno name)
[id] SEND portnum	Server provided port for scanning (from /PORT)
[id] SEND	Server is ready to receive data inline (from /DATA)
[id] SCANNING filename	File scan started
[id] INFECTED virusname : filename : message	File is infected
[id] DONE viruscount : filename	File scan is complete
[id] DONE viruscount : filename : SHA1-Hash	File scan is complete (for SCAN/FILE-SHA1)

ERRORS

A specific subset of errno names may be returned to indicate errors

EINVAL	Invalid argument
ENOTSUP	Argument or option not implemented
EACCES	File or directory could not be read
ENOENT	File or directory does not exist; no connection to port
EAGAIN	Server temporarily out of resources
EFAULT	Server internal error

EXAMPLES

Client connects to server port
Server accepts the connection

```
|                  READY
|SCAN/FILE /tmp/foo
|                  | QUEUED
|                  | SCANNING "/tmp/foo"
|                  | DONE : "/tmp/foo"
|QUIT
|                  | OK
```

Client closes the socket
Server closes the socket

```
Client connects to server port
Server accepts the connection
                        READY
     |ENABLE RECURSE 1
     |                    | OK
      |1 SCAN/FILE /tmp/foo.zip  |
      |                    | 1 QUEUED
      |2 SCAN/FILE /tmp/foo.exe  |
      |                    | 2 QUEUED
      |                    | 1 SCANNING "/tmp/foo.zip" -> "/some/file"
      |                    | 2 SCANNING "/tmp/foo.exe"
      |                    | 1 SCANNING "/tmp/foo.zip" -> "/other/file"
      |                    | 1 INFECTED CVDL W32/Sick : "" -> "/other/file" : end at 1234
      |                    | 1 DONE : "/tmp/foo.zip"
      |QUIT                  |
      |                    | OK
      |closes the socket
```

Error Reports

Please report all bugs to support@cyber.com Make sure to include the version of vfindd, the platform and OS, the script or command used, the complete output showing the bug, a short description of the problem, and contact information.

SAMPLE SOURCE CODE

The following sample source code in the C language is intended to be a starting point for anyone who want to implement the SVSP protocol with the VFindD.

```
#include <time.h>
#include <stdio.h>
#include <errno.h>
#include <ctype.h>
#include <fcntl.h>
#include <string.h>
#include <limits.h>
#include <stdlib.h>
#include <unistd.h>
#include <alloca.h>
#include <sys/types.h>
#include <sys/stat.h>
...
...
```

```
...


#ifdef CONF_OS_WIN32
#define LOCALHOST "127.0.0.1"
#else
#define LOCALHOST "localhost"
#endif
...
...

int
vf_init(const char *const *argv)
{
    ...
    ...

    if (!servers_connected())
        connect_server(NULL, LOCALHOST);

    if (!servers_connected()) {
        printf("##==>>> No server connected -- exiting\n");
        fflush(stdout);
        exit(1);
    }

    ...
    ...

    return 1;
}



   ...
   ...
   ...

struct queue_data {
    struct queue_data *next;
    int pending_requests;
    const struct vfconf *config;
    int result;
    struct vfstat *vfstat;
    int sock;
    void *user;
};
```

```
struct per_request {
   const char *filename;
   struct queue_data *queue_data;
};

static int
handle_scan(int server,
          char *text,
          int size,
          int id,
          void *ptr)
{
   char buf[BUFSIZ];
   struct per_request *request = ptr;
   struct queue_data *q = request->queue_data;
   char *filename, *virus, *message;
   int bytes, error = 0;
   int waiting = 0;
   int file;
   char *args;
   char *end;
   int count;
   long port;
   int channel;

#ifdef DEBUG_CLIENT
   fprintf(stderr, "HANDLE %d: '%d %s'\n", size, id, text);
#endif

   args = nextword(text);
   text = trim(text);
   args = trim(args);

   if (q->pending_requests <= 0) {
      unexpected("SCAN", "No pending requests", NULL);
      goto done;
   }

   /* FIXME: break up in separate functions ? */

   if (strcasecmp(text, "QUEUED") == 0) {
      set_available(server);
      return 0;
   }

   if (strcasecmp(text, "SCANNING") == 0) {
      ++q->vfstat->total_file_count;
```

```
        announce_check(q->sock, args, q->user);
        return 0;
    }

    if (strcasecmp(text, "DONE") == 0) {

        if (split3(&virus, &filename, NULL, args)) {
            count = (int) strtoul(virus, &end, 10);
            if (count < 0 || *end != '\0') {
                unexpected("SCAN", text, virus);
                count = 0;
            }
            announce_done(q->sock, filename, count, q->user);

            if (q->vfstat != NULL) {
                q->vfstat->total_virus_count += count;
            }
        }
        else {
            unexpected("SCAN", text, args);
        }
        goto done;
    }

    if (strcasecmp(text, "INFECTED") == 0) {
        if (split3(&virus, &filename, &message, args))
            announce_found(q->sock, filename, virus, message, q->user);
        else
            unexpected("SCAN", text, args);

        return 0;
    }

    if (strcasecmp(text, "SEND") == 0) {
        if (args == NULL) {
            unexpected("SCAN", "SEND: missing port (NYI)", args);
            goto done;
        }

        port = strtoul(args, &end, 10);
        if (*end != '\0' || port < 0 || port != (unsigned short) port) {
            unexpected("SCAN", "SEND: bad port", args);
            goto done;
        }
#ifdef DEBUG_CLIENT
        fprintf(stderr, "Delivering '%s' to port %d\n",
                    request->filename,
```

```
                (int) port);
#endif
        file = open(request->filename, O_RDONLY);
        if (file < 0) {
            perror(request->filename);
            goto done;
        }

        for (;;) {
            error = channel = open_channel(server, (int) port);
            if (channel >= 0)
                break;

            if (!is_temporary(error)) {
                printf("##==>>> SCAN channel failed: %s\n", strerror(-error));
            }
            if (communicate() == 0) {
                if (option.quiet < 1 && !waiting++)
                    printf("##==> Waiting for response from server\n");
                throttle();
            }
        }

        for (;;) {
            bytes = read(file, buf, sizeof buf);
            if (bytes < 0) {
                printf("##==>>> SCAN read failed: %s\n", strerror(-error));
                error = bytes;
                break;
            }
            if (bytes == 0) {
#ifdef DEBUG_CLIENT
                fprintf(stderr, "Delivered '%s' to port %d\n",
                        request->filename,
                        (int) port);
#endif
                error = 0;
                break;
            }

            error = send_command(channel, buf, bytes);
            if (error < 0) {
                printf("##==>>> SCAN channel SEND failed: %s\n",
                    strerror(-error));
                break;
            }
            if (error < bytes) {
```

```
            unexpected("SCAN", "SEND", "not all data received");
            break;
         }
         if (communicate() == 0)
            break;
      }
      close(file);
      if (error < 0)
         goto done;

      error = close_channel(channel);
      if (error < 0)
         goto done;

      return 0;
   }

   error = errmsg(text);
   if (error < 0) {
      printf("##==>>> SCAN command failed: %s, %s\n",
            strerror(-error),
            args);
      goto done;
   }

   unexpected("SCAN", text, args);


done:

 ...
 ...
 ...

   if (request != NULL) {
      if (request->filename != NULL)
         free((char *) request->filename);
      free(request);
   }

   count = --q->pending_requests;
#ifdef DEBUG_CLIENT
   fprintf(stderr, "REQUEST COMPLETE, %d PENDING\n", count);
#endif

 ...
 ...
```

```
    return count >= 0;
}


static int
scan_request(const char *filename, int status, void *data)
{
    int use_local = option.localhost;
    struct queue_data *q = data;
    const struct vfconf *config = q->config;
    struct per_request *request = NULL;
    const char *failure = NULL;
    int id, server = -1;
    int waiting = 0;
    int error;

    if (filematch(filename, config->noscans)) {
        if (option.quiet < 1)
            printf("##==> Not scanning: \"%s\"\n", filename);
        return 0;
    }

    if (status < 0) {
        error = status;
        failure = filename;
        goto fail;
    }

    if ((status & WALK_LINK) &&
        ((!config->follow_symlinks && (status & WALK_DIR)) ||
         (config->ignore_symlinks && (status & WALK_FILE)))) {

        if (option.quiet < 1)
            printf("##==> Not scanning: \"%s\"\n", filename);
        return 0;
    }

    if (use_local && filename[0] != '/') {
        if (option.quiet < 1)
            printf("##==>> Ignoring --localhost for relative filename\n");
        use_local = 0;
    }

    if ((status & WALK_DIR) && !use_local) {
        return 1;       /* Nothing more to do, all handled inside walk() */
    }
```

```
error = server = select_server();
if (server < 0) {
    failure = "select_server()";
    goto fail;
}
set_busy(server);

error = set_options(server, q->config);
if (error < 0) {
    failure = "set_options()";
    goto fail;
}

/* free() called in handle_scan()
 */
request = calloc(1, sizeof *request);
if (request == NULL) {
    error = -ENOMEM;
    failure = "vf_scan()";
    goto fail;
}
request->filename = strdup(filename);
if (filename == NULL) {
    error = -ENOMEM;
    failure = "vf_scan()";
    goto fail;
}
request->queue_data = q;

for (;;) {
    id = request_id(server, request, handle_scan, &q->result, "SCAN");
    if (id < 0 && is_temporary(id)) {
        if (communicate() == 0) {
            if (option.quiet < 1 && !waiting++)
                printf("##==> Waiting for response from server\n");
            throttle();
        }
        continue;
    }
    if (id < 0) {
        error = id;
        failure = "request_id()";
        goto fail;
    }
    break;
}
```

```
    error = send_commands(server,
                          "%d SCAN/%s %s\r\n",
                          id,
                          use_local ? "FILE" : "REQUEST",
                          filename);
    if (error < 0)
        goto fail;

    ++q->pending_requests;
#ifdef DEBUG_CLIENT
    fprintf(stderr, "NEW REQUEST, %d PENDING\n", q->pending_requests);
#endif

    return !(status & WALK_DIR);        /* SCAN/FILE recurses on the server */


 fail:
    if (server >= 0)
        set_available(server);

    if (request != NULL) {
        if (request->filename != NULL)
            free((char *) request->filename);
        free(request);
    }
    if (failure != NULL) {
        errno = -error;
        perror(failure);
    }
    return error;
}


...
...
...


static struct queue_data *queue = NULL;

...
...
...

int
vf_queue(const struct vfconf *config,
```

```
        int sock,
        const char *filename,
        FILE *file,
        long size,
        int type,
        struct vf *vf,
        struct vfstat *vfstat,
        void *user)
{
   int error;
   struct queue_data *next;

   if (queue != NULL && queue->pending_requests <= 0) {
      while (queue->next != NULL && queue->next->pending_requests <= 0) {
         next = queue->next;
         queue->next = next->next;
         free(next);
      }
   }
   else {
      next = queue;
      queue = calloc(1, sizeof *queue);
      if (queue == NULL)
         return -ENOMEM;
      queue->next = next;
   }

   queue->config = config;
   queue->vfstat = vfstat;
   queue->sock = sock;
   queue->user = user;

   switch (type) {

   case VF_FILE_SCAN:
      error = process_file(file, filename, config, queue);
      break;

   ...
   ...
   ...
   ...

   default:
      error = -EINVAL;
   }
```

```
    if (file != NULL)
        fclose(file);

    return error;
}

...
...
...
```

SVSP Frequently Asked Questions

1) I verified that the data that I sent to the vfindd socket was read using truss, however vfindd didn't seem to process all of the data. Sometimes sending more commands to vfindd would cause the unprocessed commands to process. A work around is to send a new line, which causes the commands to be processed.

We recognized that the situation happens when a user uses telnet to send multiple scan-requests to vfindd at the same time, but it does not happen when users use vfindc (vfind-client) which is provided as the default client. The source code example provided with this document is from the working vfindc program.

The cause of this situation is if you try to send multiple SVSP-commands on one request. Basically, VfindD handles the requests from a client using event (request)-driven-approach to optimize its performance for scanning and reduce some additional loads, which are not directly, related its scanning operation. The VfindD program uses a new-line ("\r\n" or "\n") as its parsing delimiter to recognize a valid SVSP-command in the stream. In other words, the scan-procedure (or any responses) on VfindD is invoked only when a new request from a client has arrived on the socket.

If a request arrives at VfindD through a socket, VfindD reads all the data on the socket and save it to its queue. VfindD then picks-up one request from the queue to start scanning. This situation is good under the situation when one request has one SVSP-command. Of course, a client can send multiple requests without any problem if each request has one command but if a client tries to send multiple SVSP-commands in one request, you may meet a "buffered scanning situation". Since one request will invoke one scan even though the remaining commands are still in the queue, the second command which was in the first request may need to wait for the next new request to be run in a FIFO manner.

We recommend that a client sends one command per request but you can send multiple requests on the same socket. This approach was already implemented in vfindc (vfind-client) and it can be easily implemented by users in different programming languages. The only exceptional case in which this approach may not be easy is the situation if a customer uses telnet to send files/directories to VfindD. That is because the "telnet" requests users to input all data manually and there is no room for an user to implement the logic/approach recommended above. In this case, we recommend that an user gathers all the files in one directory and send the directory name as one request. Following are examples of two possible cases of a "buffered scanning situation". This example is for one request with five commands.

[1] Case #1

Added a new-line ("\r\n"-> Pressing "Enter" when editing a request) at the end of every command when he/she edits the ONE request as follows.

SCAN/FILE /dummy/file1.txt
SCAN/FILE /dummy/directory1
SCAN/FILE /dummy/file2.txt
SCAN/FILE /dummy/directory2
SCAN/FILE /dummy/file3.txt

- ⇨ *Operations on vfindd*
- ⇨ *When the ONE request (including five commands) arrives at vfindd (socket), vfindd reads all the data in the socket. All five commands (as a ONE request) were saved in the daemon-queue, and the event-handler invokes scan-process for the request in the queue. After parsing the request using the new-line delimiter ("\r\n" or "\n"), the first command is recognized as a formal command from the client and started its scanning. The other four commands are still in the queue and wait for the next request from the client. The next request may be a formal command or just new-line ("Enter"), and the commands in the queue start scanning (eventually) on a FIFO-basis. The request to invoke the event-handler should be from a client.*

[2] Case #2

No new-line ("\r\n") at the end of every command when he/she edit the ONE request as follows.

SCAN/FILE /dummy/file1.txt SCAN/FILE /dummy/directory1 SCAN/FILE /dummy/file2.txt SCAN/FILE /dummy/directory2 SCAN/FILE /dummy/file3.txt

=> Operations on vfindd

Vfindd sends the "ENOENT" error message to the client, which means "The file or directory does not exist", and wait for the next valid request on the socket. That's because, there is no parsing delimiter (new-line) at the end of each command, (after reading and parsing the data) vfindd considers the data/stream in the request as follows.

- "SCAN/FILE" ==> The formal (and first) SVSP command, and
- All the remaining data after the "SCAN/FILE"==> file/directory name. It's because there is no newline at the end of one command. Of course, there is no file/directory-name like "/dummy/file1.txt ... SCAN/FILE /dummy/file3.txt", and vfindd sends "ENOENT" message to the client, which means "File or Directory does not exist."

This answer was tested using VSTK-T Version 179.

2) VfindD misses QUEUED responses.

Unlike the "SCANNING","DONE" or "Exx.. (Error names)", the "QUEUED" response from vfindd-mt MAY not be sent to the client especially if you use the multi-threading option with many threads and many files. So, do not use the "QUEUED" response as your trigger index for another action in your application.

The reason is that (under multi-thread environment) the thread for adding target files to the internal scan-queue and the thread(s) to pop-up the task from the queue (and start the scan a file) are different threads. After target files are added to the scan-queue by the main thread, other new created threads try to pop-up each task from the queue and start scanning. The order of adding the files to the scan-queue is independent of the order for pop-up (and scanning) the files by new thread by the operating systems kernel's thread scheduler.

The main thread takes the filenames and adds them to the scan-queue. Then it does some internal logging procedure, and then tries to send the "QUEUED" response to the client if the task (file) in the scan-queue is still in the queue, in other words, it has not started scanning.

In most cases, especially if the number of target files is big (which means the number of newly created threads is big), the "QUEUED" message is sent to the client to inform that the task/file was queued for scan, however in some cases, the "QUEUED" message may not be sent to the client, because the queued task/file was popped quickly by another thread and started scanning BEFORE the main thread could check if the task is still in the queue. When the main thread checks the status, if the task is already popped and started scanning, the main thread does not send the "QUEUED" message because the file is not in QUEUE, it is in process. This answer was tested using VSTK-T Version 179.

3) Sending bad commands or random data causes vfindd to crash.

VfindD sends an "EINVAL" message to the client, which means "Unrecognized command", and waits for the next request. We have not been able to cause it to crash using bad commands or random data. Example:

 # telnet 127.0.0.1 8081 (==> vfindd-mt on port 8081)
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.
READY
wrongCommand (==> unimplemented command)
EINVAL Unrecognized command: wrongCommand (==> EINVAL from vfindd-mt)
SCAN/FILE /opt/vstk/README.txt
QUEUED
SCANNING "/opt/vstk/README.txt"

DONE 0 : "/opt/vstk/README.txt"

As you can see from this test we are not able to cause vfindd to crash by sending it bad commands. Further tests show that sending random data also does not cause it to crash. We are not aware of this problem with any version of vfindd, however if you are using a version prior to VSTK-T 179 you are using an unsupported obsolete version. Please upgrade. This answer was tested using VSTK-T Version 179.

4) If the vfindd process receives a request to scan a nonexistent file the client request to close the connection is not handled gracefully.

VfindD sends a "ENOENT" message to the client, which means "File or Directory does not exist", and waits for the next request. Example:

 # telnet 127.0.0.1 8081
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.
READY
SCAN/FILE nonexist
QUEUED
ENOENT "nonexist" (==> ENOENT from vfindd-mt)

As you can see from this test we are not able to cause vfindd to fail in any way by sending it a request to scan a nonexistent file. This answer was tested using VSTK-T Version 179.

5) VFindd appears to create Zombie threads.

All the threads generated from the process (vfindd-mt) are terminated if the process (vfindd-mt) is terminated. Please check if you invoked another vfindd-mt on different port(s) and they are not terminated. This answer was tested using VSTK-T Version 179.

6) How many copies of the vfindd daemon can I run at the same time?

As many as your system has resources for. You can run multiple VfindD(s) on different ports with "--SVSP-port=" option. Without this option, VfindD uses 8081 as its default port to listen the SVSP-connection. This answer was tested using VSTK-T Version 179.

7) Are there other APIs beyond SVSP?

Yes, but SVSP is the one that we recommend. We also support APIs that are unpublished for specific customers and we support the ClamAV API for use of vfindd with open source programs that conform to the ClamAV API.

8) What is the difference between SCAN/REQUEST and SCAN/FILE?

The SCAN/REQUEST command was designed for a situation when the server (vfindd) is on a remote machine. In other words, the "SCAN REQUEST" command is for when the files to be scanned and the VfindD server are on different machines. While the "SCAN/REQUEST" command can be used for a client on the same machine it causes VfindD to perform additional work which is not necessary for "local' files. If the client is on the same machine with VfindD, we recommend customers to use "SCAN/FILE" command, which does not consume additional hard-disk resources.