# Guide to Safe Gophering

**Paul Lindner**

In any software product as large and complex as gopher there is the possibility of security holes. In this paper we'll show you what they are and how to run a secure gopher server, gopher client, and gopher public access system.

**April 18, 1994**

## 1.0 Introduction

Gopher has had its small share of security problems. The University of Minnesota, in conjunction with CERT and other computer security agencies attempts to keep the Gopher software as secure as it possibly can.

Now that the gopher software is installed on thousands of sites it is imperative that all holes be documented and fixed. We believe that release 1.13 and 2.013 are free of security holes and that you should upgrade to these releases as soon as possible.

We make no claims as the security or insecurity of other software. This paper only applies to the Unix/VMS Gopher Software released by the University of Minnesota.

## 2.0 The Origin of Holes

There are a few programming practices and design decisions that have caused most of the holes in gopher software. These practices pervade other software packages too.

### 2.1 Use of software routines to insure security.

The most popular method of maintaining a secure environment is to use the chroot() system call to change the location of the / directory to a subdirectory. This is the method used by most FTP servers. Consider the following code snippet:

```
chroot("/usr/local/gopher-data");
```

This code causes the program to instantly think

```
/usr/local/gopher-data
```

is the slash directory. Thus if you try to access /etc/passwd you will, in fact, be attempting to access

```
/usr/local/gopher-data/etc/passwd
```

There are a couple of problems with this scheme however. The first is that you need root privileges to use this system call. Also, sometimes you *want* to access files that aren't in your gopher-data directory. For instance you might want to make a symbolic link to your manual pages like this:

```
ln -s /usr/man /usr/local/gopher-data/manual-pages
```

The solution to both of these problems was contributed by John Sellins from the University of Waterloo.

Instead of relying on the chroot() system call, a suite of software routines were written to insure that all requests are legitimate. The routines strip out all occurrences of '..' in gopher requests and append the server data directory to certain requests. If these checks weren't in place a cracker could do this:

```
0/../../../../../../../../../../etc/passwd
```

This request would retrieve the password file of the system running the gopher server if it wasn't for the dot stripping routines.

Our example of a symbolic link works however, since there aren't any '..' character strings in the request.

As we'll see later, bugs and oversights in this section of code have caused security holes.

### 2.2 Use of Dangerous Routines in the Server and Client

The server and client use two highly dangerous routines to implement portions of the code. These calls are system() and popen(). The server uses this code to decompress files on the fly and run shell scripts to perform functions not implemented in the server. The client uses these routines to execute 'helper' applications to send email, invoke telecommunications download protocols and display many data types.

The problem with both of these calls is that they use the shell 'sh' to evaluate the expression sent to them. The shell has a very extensive feature set that makes it difficult to ensure security through software routines. It's possible for a malicious user to exploit these weaknesses to actually run commands on the server. For instance the gopher server has a special notation for running an external shell script. Here's an example:

```
exec:arg1 arg2:/bin/legisearch
```

In this case the server executes:

```
popen("/bin/legisearch arg1 arg2");
```

which in turn executes:

```
sh -c "/bin/legisearch arg1 arg2"
```

In earlier versions of the server a malicious user could send the following command to a server running with the '-c' (non-chroot) option:

```
exec:arg1 ;cat /etc/passwd:/bin/legisearch
```

The server then would then execute:

```
sh -c "/bin/legisearch arg1 ;cat /etc/passwd"
```

The semicolon (;) tells the shell to execute a *second* command and add the results to that of the first. In this case the command prints out the contents of the Unix password file.

This problem will only occur if you have shell scripts on your server (the software will not execute non-existent shell scripts.) If you're using the default chroot() mode a cracker won't be able to get at files outside of the gopher directory tree. However, they could still wreak major damage to your gopher data.

This problem of hidden code execution plagued earlier versions of the client as well. The Gopher client allows connections to telnet sites by launching the telnet command with the name of a host. It does this by using the system() system call like this:

```
system("telnet pubinfo.ais.umn.edu");
```

However if an unscrupulous server administrator created a bogus telnet item with a hostname like this:

```
"pubinfo.ais.umn.edu;echo + >.rhosts"
```

This would cause the .rhosts file of the person using the client to be compromised when they selected the item. Gopher links like this are 'telnet bombs' just waiting to be opened, much like physical mail bombs.

Starting with release 2.0.12 we've started replacing these routines with more general secure code that does not use the shell. So far we've converted most of the client side and some of the server calls. Eventually we will be rid of this problem altogether.

## 3.0 Known Holes

Here is a list of the known holes in gopher and gopherd for Unix system. It is probably not complete.

- All releases between 0.9 and 1.12 and 2.0 to 2.04 may allow executing external commands with the exec: facility.
- All releases between 2.0 and 2.012 and before 1.13 may allow ‘telnet bombs’.
- Releases before 2.0.12 had dedot routines that could be tricked with the right combination of quotes and dots.

## 4.0 Known Bad Practices

These aren’t holes really, but they are things that people do that could prove very, very dangerous..

Some sites allow simultaneous ftp and gopher access to the same set of files. Often a directory will be writable for uploads for ftp users. If the execute bit is turned on for uploads the cracker could upload shell scripts to the server, which could then be activated by a gopher client.

## 5.0 Insuring Security of the Gopher Server

Even with the potential for security holes it is still possible to run a secure gopher server. The following steps are insurance against future security holes.

The safest thing in any circumstances is to not run the server at all. If you are very security conscious you may not want to trust your data to software that doesn’t come with a warranty.

However this is short sited at best. There are a number of techniques to make your gopher server secure.

- Avoid the chroot() (-c) option if you can. This is perhaps the most dangerous portion of the gopher code. It’s been gone over with a fine tooth comb, but there’s no guarantee that we’ve closed every loophole.
- Always, always, always run the server as a non-root user i.d. The server has a -u parameter to set the username. Use it! Any damage that a user can cause can be minimized by running as a non privileged user.
- Be careful with shell scripts. The gopher server can’t save you from shooting yourself in the foot by using a poorly written, insecure shell script.
- Always log your transactions! Use the -l option of gopherd to specify a log file like this.

  ```
  -l /usr/adm/gopherlog
  ```

  This will give you an audit trail in case anything happens. It also gives you a way to find anomalies in access. A log analyzer will be able to tell you if you’re receiving a disproportionate number of connections from certain sites
- If your server contains sensitive information you may want to use the ip/hostname security option of the server. This is done by using ‘access:’ lines in your gopherd.conf configuration file.

  The following lines will allow only hosts from the University of Minnesota to access your server. Note that this style of security has some obscure holes. If your DNS server is compromised, then so is your gopher server if it uses this form of authentication.

  ```
  access: default !read,!browse,!search,!ftp
  access: .umn.edu read,browse,search,ftp
  ```
- Consider executing your scripts with ‘taintperl’. This version of perl makes sure that you never use user input to execute a command on your system.

## 6.0 Insuring the Security of Your Client

The client is less problematic that the server, but it still has it’s own problems.

- Upgrade to 2.013 or higher. Earlier versions could potentially execute a ‘telnet bomb’. If you read the warning screen you’ll probably notice it, but you could easily miss it.
- Certain file types are inherently insecure. You should trust the source of postscript and Microsoft Word documents before viewing them. Both systems can execute system macros that could alter data and files on your system. This is not a gopher problem, but a general information retrieval problem. Other file formats can potentially cause the same problems.

## 7.0 Insuring the Security of Your Public Access Client

A public access client allows people to connect to the internet, (often without a password) and use a gopher client. This allows dial up users and those without the resources to run a gopher client on their own system access to the gopher network. A public access client can be an enormous security risk if not installed properly. The following steps should give you a secure public access client, but again, we make no guarantees.

The first step is to create the home directory of the for the guest account. This directory should not be writable nor owned by the guest account.

In this directory you will want to place a shell script that will start the gopher client automatically. Here is an example:

```
#!/bin/sh

GHOME=/home/hafnhaf/gopher

USER=`whoami` export $USER

eval `tset - -s -m "network:?vt100" -m
"unknown:?vt100" -m "dumb:?vt100" -m
":?vt100"`

echo "I think you're on a $TERM terminal"

sleep 3

PATH=$GHOME/bin

export PATH

exec /home/hafnhaf/gopher/gopher -s $GHOST 70
```

Note that we set the path to a bin directory inside of the guest account directory. We want to insure that only programs inside of this directory are run. The reason we do this is that many programs allow you to get to a shell, like telnet, tn3270, less, more and many others. You should put secure versions of these programs in the bin directory inside the guest directory.

Also note that we use the -s option for gopher. This invokes the 'secure' option of the client. In this mode the client won't let you save or print.

Finally you can create the guest account. Make sure that the shell is specified as the script. Don't put /bin/csh or /bin/sh for the shell. Here's an example entry for a guest account:

```
gopher::10000:10:Public Access Account:/home/
gopher:/home/gopher/Startup
```

Don't forget to honor the warnings for the regular client as well. On public access systems people will actually *try* to run telnet bombs.