Handling Passwords with Security and Reliability in Background Processes

Don Libes – National Institute of Standards and Technology

ABSTRACT

Traditionally, background automation of interactive processes meant giving up security and reliability. With the advent of software such as Expect for controlling interactive processes, it has become possible to improve reliability and security with relative ease.

This paper reviews the reliability aspects but focuses primarily on the security aspects, presenting several non-obvious techniques for dealing with passwords and other sensitive information in background processes. These techniques require no changes to existing programs and no new security systems are necessary. With the appropriate tools and examples, these techniques can be applied with surprisingly little effort to a wide variety of problems.

Introduction

Shell scripts cannot automate interactive processes except in the simplest of ways. In particular, data can be written to a process but only following one path through the program. Responding to programs is not possible. Problems such as timing and buffering can make automation difficult if not impossible.

It is possible to reliably automate interactive processes with a variety of tools including C, Perl, and Emacs. For simplicity, I will present examples in Expect [Libes94], but other tools are similarly applicable. Indeed, both Perl's "chat2" [Schwartz90] and the C/C++ Expect library were modelled after the Expect program.

Automating **`ftp`** is a common problem. The usual solutions are to use an **`.ftprc`** file or an in-line "**`<<`**" script. Unfortunately, these sacrifice both reliability and security. Reliability is lost because these mechanisms offer no way to verify that the commands succeed. Security is lost when passwords are stored as cleartext in a file or passed as cleartext through command-line arguments. (For simplicity, from now on I will refer to all sensitive information as "passwords".) Security systems such as Kerberos [Miller87] do not address these problems.

This paper does not address the simple cases where applications are entirely under your control and can be modified or otherwise forced to run without passwords. **`sudo`** [Nieusma] and similar programs provide a direct solution to these problems.

In contrast, the problems addressed by this paper *demand* a password. A simple case might be that of designing a means to use a service from a commercial provider in the background. An automated solution requires you to log in and supply the password. The commercial service is not under your control.

This paper describes several techniques that can be used to handle passwords in background processes in a secure way. The techniques are non-traditional yet relatively simple to implement. These techniques will be demonstrated using Expect.

Expect – An Overview

Because the examples in this paper are written in Expect, an overview of the language is provided here. The implementation and philosophy of Expect is described at length in the literature [Libes90, Libes91]. Briefly, scripts are written in an interpreted language. Commands are provided to create interactive processes and to read and write their output and input. Expect is named after the specific command which waits for output from a program.

The language of Expect is based on Tcl [Ousterhout94]. Tcl is actually a subroutine library, which becomes embedded into an application and provides language services. The resulting language looks very much like a typical shell language. There are commands to set variables (**`set`**), control flow (**`if`**, **`for`**, **`continue`**, etc.), and perform the usual math and string operations. Of course, UNIX programs can be called (**`exec`**). All of these facilities are available to any Tcl application. Tcl is completely described by Ousterhout.

Expect is built alongside of Tcl and provides additional commands. The **`spawn`** command invokes a UNIX program for interactive use. **`send`** sends strings to a process. **`expect`** waits for strings from a process. **`expect`** supports regular expressions and can wait for multiple strings at the same time, executing a different action for each string. **`expect`** also understands exceptional conditions such as timeout and end-of-file.

Using Expect it is possible to script **telnet**, **ftp**, **rlogin**, **rz/sz**, and numerous other programs. Many of these tasks fall in the domain of system administration. For example, a system administrator creating thousands of accounts each semester will find an automated **passwd** program much more convenient than having to type in each password manually.

The following script is another example, driving the **fsck** program so that one class of questions is answered "yes" while another is answered "no". If anything else appears, control is temporarily turned over to a user to answer it.

```
while 1 {
    expect {
        eof                     {break}
        "UNREF FILE*CLEAR\\?"  {send "y\r"}
        "BAD INODE*FIX\\?"     {send "n\r"}
        "\\? "                  {interact +}
    }
}
```

Using a script like this one can substantially raise the reliability of tasks that normally require interactive use.

Expect and related programs can be put to a wide variety of uses as others have found [Woodson91, Morrison92, Stevens92, Caffrey92, Dichter93] solving problems which were not even recognized as problems only because there were no good solutions.

A particularly common problem addressed by interaction automation software is entering passwords. Passwords are usually entered by hand. Most programs (**rlogin**, **crypt**, etc.) use **getpass**, a UNIX library function, which reads the password from **/dev/tty**. Since **/dev/tty** cannot be redirected from the shell, the user must enter keystrokes manually. A variety of kludges have appeared over the years to defeat such security measures. Why? Because entering passwords manually is tiresome. Consider having to enter the same passwords every day to make use of a service.

The remainder of this paper will focus on automating the handling passwords with special regard to background processes. Background processes are a general goal – if you can run a process in the background, it is completely automated. You can turn your attention to other things.

In many cases, everything in a process can be automated except for the password entry. Were this automated, the process as a whole could be made into a background process. So how do we fix this problem?

I will describe several common scenarios involving handling passwords. In each case, I will explain how to automate the handling, usually resulting in a completely automated and backgroundable process.

I will use the term *script* to refer to that which performs the automation and may indirectly run the true *program* of interest. Of course, the program may indeed be a script. Similarly, the role of the script may be played by a compiled program. However, the terms I will use are accurate for most applications.

Technique 1: In the Foreground, Prompt For Passwords Ahead Of Time

The technique described in this section is appropriate for a user who decides at the spur of the moment to schedule a background task for a later time. (*Spur of the moment* is not meant to imply the command is trivial or light-hearted. Virtually all interactive commands are spur of the moment.) For example, imagine a user wants to automate a **telnet** session to another host. The session must occur several hours later, however. The user will not be present to supply the password.

One solution is to write a script that prompts for the password immediately when the user makes the request. The script begins running interactively. The first thing it does is prompt for passwords. Once all sensitive information has been gathered, the script disconnects from the terminal and continues in the background, perhaps sleeping if necessary until an appropriate time. The script then starts the program, interactively answering the program's requests for passwords.

Below is a sample of such a script using Expect. The script is not setuid and may be readable to others since no passwords are embedded within.

```
# prompt and collect password for later
stty -echo
send "password? "
expect -re "(.*)\n"
send "\n"
set password $expect_out(1,string)

# got the password, now go
# into the background
if {[fork] != 0} exit
disconnect

# now in background, sleep (or wait
# for event, etc)
sleep 3600

# now do something requiring the password
spawn rlogin $host
expect "password:"
send "$password\r"
. . .
```

This script can be extended as necessary. For example, the task might **telnet** to multiple hosts or a additional hosts from the first **telnet**. Each of these in turn requires more passwords. These can be prompted for and collected when the script has begun.

The prompt should make clear what the passwords are for. It may be helpful to explain why the password is needed, or that it is needed for later. Consider the following prompts:

```
send "password for $user1 on $host1: "
send "password for $user2 on $host2: "
send "password for root on hobbes: "
send "encryption key for $user3: "
send "sendmail wizard password: "
```

It is a good idea to force the user to enter the password twice. It may not be possible to authenticate it immediately (for example, the machine it is for may not be up at the moment), but at least the user can lower the probability of the script failing later due to a mistyped password.

```
stty -echo
send "root password: "
expect -re "(.*)\n"
send "\n"
set passwd $expect_out(1,string)
send "Again:"
expect -re "(.*)\n"
send "\n"
if {[string compare $passwd \
          $expect_out(1,string)] != 0} {
    send "mistyped password?"
    exit
}
```

You can even offer to display the password just typed. This is not a security risk as long as the user can decline the offer or can display the password in privacy. Remember that the alternative of passing it as an argument allows anyone to see it if they run **ps** at the right moment.

Even without the **disconnect** command, this is a valuable technique. For example **passmass** is an Expect script that changes passwords on multiple machines simultaneously. This is useful if you have accounts on several machines that do not share password databases yet you want to use the same password on all of them. While this sounds like an obvious security hole, **passmass** can actually increase security. Because **passmass** makes it so much easier to change your passwords on all your accounts, you are much more likely to change them more frequently. And by keeping them the same, you are less likely to have to resort to writing them down in places that you shouldn't. Note that **passmass** is not recommended for widely distributed sites where communications over public networks provides little defense against password exposure. Nor is **passmass** recommended for **root**, where this idea is too simplistic and additional precautions should be taken.

Technique 2: From the Background, Prompt For Passwords When Needed

This technique described in this section is appropriate for commonly occurring tasks such as those that are scheduled at boot time or are regularly scheduled through **cron**.

One solution is to write a simple script which runs the program until it requests a password. The script then tracks down a user (possibly from a list) and requests the user talk to it (using "**talk**" or "**write**"). Once connected, the script explains what it wants and why, and then asks the user for a password. The user supplies it, the script disconnects and returns to the background to continue its processing.

In the following example, the script communicates only with a single user. The script uses **kibitz** [Libes93] to communicate. **kibitz** is a **talk**-like program notable in that it allows sharing of a process (e.g., shell) between multiple users. With the **-noproc** flag, **kibitz** supports communication without a shared process.

```
spawn kibitz -noproc $user
```

Once connected, the user can interact with the Expect process or can take direct control of the spawned process. The following Expect fragment, run from **cron**, implements the latter possibility. The variable **proc** is initialized with the spawn id of the errant process while **kibitz** is the currently spawned process. When the user presses the tilde key, control is returned to the script.

```
spawn some-process; set proc $spawn_id
. . .
. . .
# script now has question or problem
# so it contacts user
spawn kibitz -noproc some-user
interact -u $proc -o ~ {
    close
    wait
    return
}
```

If **proc** refers to a shell, then you can use it to run any UNIX command. You can examine and set the environment variables interactively. You can run your process inside a debugger or while tracing system calls (i.e., under **trace** or **truss**). And this will all be under **cron**. This is also an ideal way of debugging programs that work in the normal environment but fail under **cron**. Figure 1 shows the process relationship created by this bit of scripting.

Those half-dozen lines (above) are a complete, albeit simple, solution. A more professional touch might describe to the user what is going on. For example, after connecting, the script could send an explanation such as:

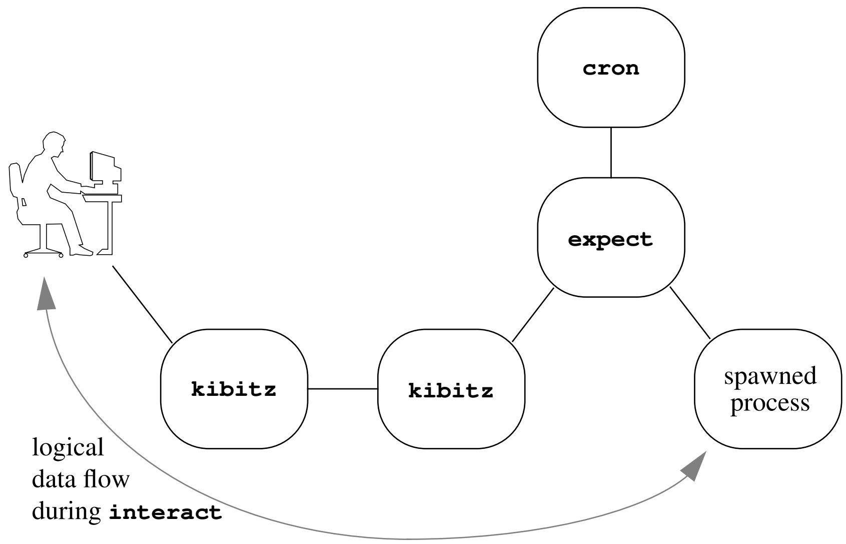

Figure 1: Process hierarchy and data flow established when Expect script running a spawned process under cron decides that it needs assistance from a user.

```
send "Host frisbee.net is requesting a
      password when I tried to login in
      as user ferdy.  Can you tell me
      what the password is (p) or should
      I let you interact (i) or kill me
      (k)?"
```

The script describes the problem and offers the user a choice of possibilities. Here is how the response might be handled:

```
expect {
    k {
        send "ok, I'll kill myself...
              thanks"
        exit
    }
    p {
        send -i $proc [get_password]
        send "thanks!"
    }
    i {
        send "press X to give up control,
              A to abort everything"
        interact -u $proc -o X return A
        exit
        send "ok, thanks for helping.
              I'll take over now"
    }
}
close
wait
# script continues from here
```

This technique can also be used for non-essential information, such as if the script has a question about what to do in a certain situation, or is performing a backup and needs another tape.

Technique 3: Protect Cleartext Passwords in Scripts by Permission

The scenario described in the remaining techniques applies when a user does not know a password but needs a service performed that requires the password. For example, mounting devices and initiating backups are typical operations that users need to perform but which require root permission on many hosts.

An obvious solution is to embed the cleartext password in a heavily-protected script. For example:

```
spawn su
expect "Password:"
send "ak3KuIO3\r"
.
.
```

Schemes to do this without root involvement are well known, such as by using setgid scripts to artificial users and groups created just for the purpose of running such scripts. However, this is difficult to make secure and impossible on some systems when using scripts. Even when using compiled programs, secure handling of passwords is tricky and prone to mishap. The storing of cleartext passwords on a public system is a bad idea. There are too many possibilities for lapses of security. These issues are described at length in the literature [Garfinkel91].

This technique is very insecure. Do not use it!

Technique 4: Protect Cleartext Passwords in Scripts by Login

It is possible to embed cleartext passwords in scripts and protect the scripts more securely than in the previous technique by placing password-containing scripts on an entirely different host (called the *admin* host from hereon), thereby avoiding file system access holes. Rather than using file system permissions, general shell access is not permitted to the admin host. Instead, each different script is run by logging in to a different account. Their are no normal user accounts – only root has access to a general-purpose shell on the admin host.

Writing such a login script to provide a service takes little extra skill than writing any script. Programmers must avoid the obvious pitfalls such as allowing users to invoke a shell or write arbitrary files. However, these are a small subset of the usual concerns in writing setuid scripts. For example, without a shell, users can not change the **IFS** definition or play games with symbolic links.

The key concept here is that scripts can literally store passwords in them with no fear of them being exposed. They cannot be exposed because users cannot read them. They cannot read them because they cannot even log in to the machine in any but extremely restricted ways.

With this technique it is possible to write scripts that log in and connect to other machines which require passwords. For instance, a user may indirectly connect back to their own machine. Imagine a user is working late and wants to suspend the automatic backups that normally run every night at 3AM. The user logs in to the admin host as, say, "**backup-suspend**". The login script for **backup-suspend** logs into the user's host as **root** and suspends the backup. The user might see this interaction:

```
lion% telnet admin-host
login: backup-suspend
Backup suspended on host lion
lion%
```

This particular interaction could be simplified by an email interface since there is nothing interactive here but one might imagine interactions that are much more complex, perhaps even popping up a window on the user's system.

An obvious drawback of this approach is that a second host is required. However, this is not a big deal because computer are cheap. Realistically, most environments have unused computers sitting idle – oftentimes shunned just because they are slow. These slow hosts are entirely suitable for this job since the significant processing occurs on the user's host after the password-containing script has logged in. Although the admin host is executing a script, the admin host is not actually doing the cpu-intensive work, the admin host is merely telling the user host what to do. The user host is where the significant work is being done.

A second drawback of this technique is that the password is made available for exposure by network sniffing. However, this is a problem for any superuser that logs in over the network.

Finally, it should be obvious that the admin machine must be physically off-limits and its backup tapes must be secure. If either of these are not the case, then obviously the machine is not a safe place to store passwords.

One may draw the analogy that this is akin to placing all of your eggs in one basket. This is quite accurate, however this is a very small basket and easy to keep watch over. Many sites have the analogy of such a basket already, but without realizing or admitting it. Indeed, sites with servers that are kept behind locked doors *are* treating their computers as such baskets.

Technique 5: Protect Cleartext Passwords in Scripts by Using Daemons

In the previous technique, the script is invoked by remotely logging in to another host. An unfortunate attribute of that technique is that some minimal interaction is hard to avoid. In particular, programs such as **telnet** will prompt for the user name. If the user is on a UNIX-like host, they can use **rlogin** which avoids the prompt for the username. If no password is demanded, the invocation is not interactive. This may seem to be a convenience, but is really a necessity when scripts are invoked by other scripts, background processes, or in other situations where the user is not conveniently available to answer the prompts.

For instance, in heterogenous environments, users can not necessarily depend on the presence of **rlogin**. The **rlogin** program simply is not available from many PCs and Macs for example.

Many programs designed to operate on the heterogenous Internet stick to the lowest common denominator for communications functionality. For example, Mosaic and Gopher are information systems that follow links of information that may lead from one machine to another.[1] The Gopher daemon does not support the ability to run interactive programs. For instance, suppose you have a **telnet** interface (using the normal **telnetd**) to a valuable resource such as a database. You can make it available through Gopher but only in an uncontrolled way. The Gopher daemon is incapable of running interactive processes itself so it passes the **telnet** information to the Gopher client. Then it is up to the Gopher client to run **telnet** and log in.

This means that the client system has to do something with the account information. By default, the Gopher client displays the information on the screen and asks users to type it back in. Besides being redundant, this interaction means that accounts and passwords are

1.While the Mosaic interface is different than Gopher, both have the same restrictions on handling interactive processes and both can take advantage of the approach described here.

necessarily exposed to users. Unfortunately, Gopher clients cannot perform interaction automation. And even if they could, the accounts and passwords would still be made available to the Gopher client. By substituting their own Gopher client, users could obtain the passwords and then interact by hand, doing things you (as the advertiser of the service) may not want.

One solution is to use the technique I described in the previous section but modified specifically to run as a **telnet** daemon. **telnet** itself does not demand any account or password, so security is entirely up to the daemon. It is possible to make a non-interactive script simply by not querying for accounts or passwords. A trivial Expect script to run a non-interactive program as a daemon takes no special adaptation. The script merely handles the passwords as before and then runs the program. The client's invocation becomes simply:

```
telnet host service
```

Unfortunately, invocation of interactive programs demands more work because **telnet** clients default to communications with rather peculiar characteristics. Characters are echoed locally and not sent until a carriage-return is entered. Carriage-returns are received by the daemon with a linefeed appended. This peculiar character handling has nothing to do with cooked or raw mode. In fact, there is no terminal interface between **telnet** and **telnetd**.

This translation is a by-product of **telnet** itself. **telnet** uses a special protocol to talk to its daemon. If the daemon does nothing special as in the case of the script that spawned the non-interactive application), **telnet** assumes these peculiar characteristics. Unfortunately, they are inappropriate for most interactive applications. For example, the following Expect script will not work correctly as a daemon:

```
spawn /bin/sh
interact
```

Fortunately, a **telnet** daemon can modify the behavior of **telnet**. A **telnet** client and daemon communicate using an interactive asynchronous protocol. An implementation of a **telnet** daemon in Expect is short and efficient. The basic idea is to make sure that the daemon is always ready to respond to **telnet** commands at all times. This is easily accomplished with an **expect_before** statement. **expect_before** provides patterns that are tested before any explicit patterns. Thus, they do not have to be repeated for each **expect** command in an interaction.

A fragment of the Expect dialogue to handle the **telnet** protocol is shown below. Variables such as **IAC** contain the relevant protocol values. The script begins by offering to do echoing instead of the local client. SGA is also offered. SGA (Suppress Go Ahead) means that communication is asynchronous instead of synchronous. The script also offers to support the terminal type.

```
send "$IAC$WILL$ECHO"
send "$IAC$WILL$SGA"
send "$IAC$DO$TTYPE"
```

The **expect_before** command defines actions for each command that can be sent from the client. For instance, the first pattern matches an acknowledgment by the client that the server should do echoing. The second pattern is similar but for SGA. The third pattern refuses requests from the client to do anything else. The last pattern matches the offer by the client to send the terminal type. In response, the daemon acknowledges by requesting that the client go ahead and send the information.

```
expect_before {
    -re "^$IAC$DO$ECHO" {
        # accept as acknowledgment
        exp_continue
    }
    -re "^$IAC$DO$SGA" {
        # accept as acknowledgment
        exp_continue
    }
    -re "^$IAC$DO\(.)" {
        # refuse anything else
        send_user \
          "$IAC$WONT$expect_out(1,string)"
        exp_continue
    }
    -re "^$IAC$WILL$TTYPE" {
        # respond to acknowledgment
        send_user \
          "$IAC$SB$TTYPE$SEND$IAC$SE"
        exp_continue
    }
```

This is not a complete definition to handle the entire **telnet** protocol, but it suffices to give the flavor of it. Indeed, there are near a dozen extensions to **telnet** and more are added frequently. Most **telnet** daemons do not handle most of the **telnet** protocol commands. A richer implementation of the protocol is shown in [Libes94].

Once the protocol handling is defined, a more typical Expect script can follow. As an example, suppose you want to let people log into another host – such as a commercial service for which you pay real money – and run a single program there but without knowing which host it is or what your account and password are. Then, the server would spawn a **telnet** (or **tip** or whatever) to the other host.

```
log_user 0          ;# turn output off
spawn telnet secrethost
expect "Username:"
send "8234,34234\r"
expect "Password"
send "jellyroll\r"
```

```
expect "% "
send "ncic\r"
expect -re "ncic\r\n(.*)"
log_user 1          ;# turn output on
                    ;# send anything that
                    ;# appeared just after
                    ;# command was echoed
send_user "$expect_out(1,string)
```

Additional protocol commands can be exchanged at any time, however in practice, none of the earlier ones ever reoccur. Thus, they can be removed. The protocol negotiation typically takes place very quickly, so the patterns can be deleted after the first **expect** command that waits for real user data.

```
expect_before -i $user_spawn_id
```

One data transformation that cannot be disabled is that the **telnet** client appends a null character to every return character sent by the user. This can be handled in a number of ways. The following command does it within an **interact** command which is what the script might end with.

```
interact "\r" {
    send "\r"
    expect_user null
}
```

Additional patterns can be added to look for commands or real user data, but this suffices in the common case where the user ends up talking directly to the process on the remote host.

Ultimately, the connection established by the Expect daemon resembles what is shown in figure 2. Notice that the usual **telnet** daemon, **telnetd**, is not part of the figure. Rather, the Expect script plays the role of the daemon. Similarly, the pty and the interactive process replace the pty and login shell normally allocated and created by the **telnet** daemon.

The daemon could then do any operation involving passwords. For instance, the daemon could **telnet** to yet another host. But in this case the user would get only what the intermediate server allowed. By controlling the dialogue from the server rather than the client, passwords and other sensitive pieces of information do not have a chance of being exposed. There is no way for the user to get information from the server if the server does not supply it. Another advantage is that the server can do much more sophisticated processing. The server can shape the conversation using all the power of Expect. Without Expect, the user has full access to the spawned interactive program.

In practice, elements of the earlier script (containing the long **expect_before** definition) can be stored in another file that is sourced as needed. For instance, all of the commands starting with the **telnet** protocol definitions down to the bare **expect** command could be stored in a file (say, **expectd.proto**) and sourced by a number of similar servers.

xinetd [Tsirigotis92], a freely-available version of **inetd** provides control on the basis of hosts/networks and time-of-day over access to the services. **xinetd** is strongly recommended over **inetd**.

Summary and Conclusion

Shell scripts and redirection are so easy to use that users ignore the fact that they provide no reliability or security when it comes to handling passwords in the background. Even users who practice "safe computer sex" in other ways, are negligent when it comes to automation of interactive processes. This paper hopes to enlighten users and save them from the holes into which they will inevitably fall if they stick to the tools and techniques of the past.

The solutions outlined here avoid the historic problems with automating interactive processes in the background. The first two techniques avoid supplying passwords from the command-line (avoiding the well-known "**ps**" hole) or from files (avoiding the "look at the backup tape" and other holes). The last two techniques store cleartext passwords in files but in such a way that they are inaccessible yet usable by normal users.

Expect-style scripting also offers the ability of reliable control over processes. Scripts can verify responses and can retry or take alternative actions upon failure or unexpected results. When dealing with users, scripts can also shape the dialogue showing users only parts of the dialogue that are appropriate, or making substitutions in what the user sees.

Expect has been available for several years, yet these techniques are non-intuitive, and for this reason, not

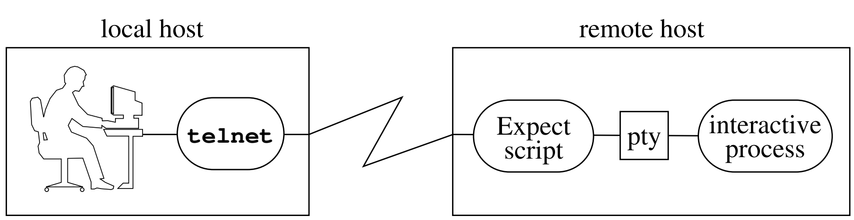

Figure 2: Expect Playing the Role of Telnet Daemon

known. This paper has shown that each of these techniques requires only a few lines of code with the result that interactive background processes can be automated with security and reliability.

All of the tools mentioned in this paper are freely available and widely portable.

Availability

Since the design and implementation of this software was paid for by the U.S. government, it is in the public domain. However, the author and NIST would appreciate credit if this software, documentation, ideas, or portions of them are used.

The scripts and programs described in this document may be **`ftp`**'d as **`pub/expect/expect.tar.Z`**[1] from **`ftp.cme.nist.gov`**. The software will be mailed to you if you send the mail message "**`send pub/expect/expect.tar.Z`**" (without quotes) to **`library@cme.nist.gov`**.

Acknowledgments

Portions of this work were inspired by Sandy Ressler and the NIST Virtual Library Project, and funded by the NIST Scientific and Technical Research Services.

Thanks to W. Richard Stevens, Henry Spencer, Bennett Todd, Miguel Angel Bayona, Brent Welch, Danny Faught, Paul Kinzelman, Barry Johnston, Rob Huebner, Todd Bradfute, Jeff Moore, and Susan Mulroney for providing suggestions that greatly enhanced the readability of this paper.

Author Information

Don Libes is a computer scientist at the National Institute of Standards and Technology where he does research related to interaction automation and occasionally logs in as root to "fix things" much to the consternation of the real system administrators there. For the development of Expect, he received the International Communications Association Innovation Award and the Federal 100 Award. He has written over 85 papers and articles as well as two books: *Life With UNIX* (co-author Sandy Ressler, publisher Prentice-Hall) and *Obfuscated C and Other Mysteries* (Wiley). He is presently working on a book called *Exploring Expect: A Tcl-Based Toolkit for Automating Interactive Programs* (O'Reilly). He can be reached at libes@nist.gov.

References

[Caffrey92] Paul Caffrey, "User Interfaces and Automating Computer Human Interaction", MSc. Thesis, Amdahl Ireland Ltd., 1992.

[Dichter93] Carl Dichter, "Surviving Software Testing", *UNIX Review*, pp. 29-36, V11, #2, February 1993.

[Garfinkel91] Simson Garfinkel and Gene Spafford, *Practical UNIX Security*, O'Reilly & Associates, Inc., June 1991.

[Libes90] Don Libes, "Expect: Curing Those Uncontrollable Fits of Interaction", *Proceedings of the Summer 1990 USENIX Conference*, pp. 183-192, Anaheim, CA, June 11-15, 1990.

[Libes91] Don Libes, "Expect: Scripts for Controlling Interactive Programs", *Computing Systems*, pp. 99-126, Vol. 4, No. 2, University of California Press Journals, CA, Spring 1991.

[Libes93] Don Libes, "Kibitz – Connecting Multiple Interactive Programs Together", *Software – Practice & Experience*, John Wiley & Sons, West Sussex, England, Vol. 23, No. 5, May 1993.

[Libes94] Don Libes, *Exploring Expect: A Tcl-based Toolkit for Automating Interactive Programs*, O'Reilly & Associates, Inc., to appear.

[Miller87] S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer, "Section E.2.1: Kerberos Authentication and Authorization System", M.I.T. Project Athena, Cambridge, Massachusetts, December 21, 1987.

[Morrison92] Brad Morrison & Karl Lehenbauer, "Tcl and Tk: Tools for the System Administrator", *1992 LISA VI Proceedings*, Long Beach, CA October 19-23, 1992.

[Nieusma] Jeff Nieusma and David Hieb, "sudo" manual page, The Root Group, Boulder, CO, undated.

[Ousterhout94] John K Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, April 1994.

[Schwartz90] Randal Schwartz, "Expect.pl", Usenet article id 1990Nov2.003228.22744@iwarp.intel.com, comp.lang.perl, November 2, 1990.

[Stevens92] W. Richard Stevens, *Advanced Programming in the UNIX Environment*, Addison-Wesley, pp. 635, 653-655, 716, September 1992.

[Tsirigotis92] Panagiotis Tsirigotis, "xinetd" manual page, University of Colorado, 1992.

1. The ".Z" file is compressed. A ".gz" version is also available which is gzipped.