Reliability of UNIX Utilities

Eric Lucas de Peslouan
Linköpings universitetet, Sweden
erilu798@student.liu.se

Muhammad Azam Akram
Linköpings universitetet, Sweden
muhak142@student.liu.se

ABSTRACT

UNIX is a very commonly used operating system, but some of UNIX utilities are not reliable. It means that these utilities sometime crash or hang. In this report we try to evaluate the reliability of some common interactive UNIX utilities. We tested different UNIX utilities by giving them random inputs and check their outputs to see whether it terminate normally or not. The important thing is that most of utilities showed significant improvement as compared to their old versions back to 1990s.

INTRODUCTION

As the usage of computer increased, problems started to raise. In modern era of technology, people emphasize on secure computer system. Operating system is one of the fundamental component of computer, so it is very important to have a reliable operating system. UNIX is a very popular one and it provides reasonably good security options. But it seems that not all of the UNIX utilities are reliable. Some of them have vulnerabilities which can lead to crashes. This weakness allows the external threats to damage the system.

Reliability of any utility depends upon how it gives output to the user, i. e. either it is the required one or an error. It is expected that a basic utility should not crashed. For our work, we divide the output of the utilities into three major categories; Proper output, hang, and crash.

Proper output: Execution of the utility ends properly and gives the required result. If an utility receives unwanted input, it should give an error message and terminate normally.

Hang: An utility can be hung or halted when it does not give any output even after taking complete input.

Crash: An utility crashes when it ends abnormally by giving unwanted results.

Researches have shown that improper inputs are one of the major reasons for why an utility does not provide the required result. Another reason for failure of the utilities is carelessness by the programmers [1].

Users do not always give right inputs to the utilities. Rather, sometimes, they can give improper input (deliberately or by mistake) which causes some unwanted outputs. For example, consider a scenario in which we have a server which is providing services to many clients. If one of the clients gives improper input to the server and as a result that server crashes, then the server will not entertain any of the future queries. This is a security problem as any attacker can attempt to give unwanted input to the server to crash it. Which results in *denial of service attack (DoS)*. DoS is an attack to a computer system that cause a loss of service to the user.

In this work, we checked the reliability of different interactive UNIX utilities which are tools that are directly interacting with the user by taking input and producing output. Reliability, in this context, means either these utilities terminate normally (with required output) or abnormally (crash or hang).

We enhanced the research work carried out by a team of researchers in 1990 [1] . They studied many utilities (with different versions) on different UNIX system. In 1995, the same researchers produced revision work of their research in 1990 [2]. During our tests we noticed that most of the utilities showed significant improvement in terms of reliability as compared to its older versions.

For testing any software, testers give different types of input to the system and check its result. We followed the same way. We tested different interactive UNIX utilities by giving them random input strings and checking their output.

We selected two types of UNIX utilities for our test work: network based and shell based utilities.

Network based utilities are the software which can be used to perform network related tasks, for example ftpd is used to send files from one host to others over the network. For test purpose we used ftpd, postfixd, sshd and apache which are very commonly used utilities. Shell based utilities are tools which the user uses to interact with the computer. For example, wc is a shell based utility which counts the number of characters in files.

We performed our tests on grep, uniq, indent and bash. We tested both network and shell based utilities against random strings of characters. For all the test we used the operating system debian GNU/Linux.

Our work follows this way:

- Generating input strings of random characters by the help of a tool called as *fuzz*
- Suppling the random input to the target utility, we want to test for reliability, by using another tool named as *ptyjig*
- Checking the response of tested utilities on supplying random inputs
- Evaluating the results

We kept ourselves confined to test the reliability of different interactive UNIX utilities and find the reasons why these utilities crash or hang. Though we did not solve or fix the problem to remove the vulnerabilities, but our method of testing finds some real bugs which result in failure of any utility.

In the METHOD section we describe the method we followed for testing different UNIX utilities, whereas, in the RESULT AND ANALYSIS section we demonstrate the outcomes and analysis of our testings. In the end we make our conclusion on the base of our work.

METHOD

In order to test the reliability of the UNIX utilities we had to setup a clear testing procedure. We decided to use a method similar to the one used in the previous studies [1]. Our method can be decomposed in the following steps :

- Generate random input
- Supply input to utilities
- Check the behavior of utilities on receiving the input

The main difference between our method and the one used in the previous studies, is in the inputs generation. We generate the input "on-the-fly" while testing, and we save only the ones which generated improper behavior, whereas the other technic consist in first generating and storing inputs in files and testing after.
To apply correctly this procedure we wrote some scripts, which perform all the task automatically.
For each test our script checks the return value of the utility to determine whether there is an error or not.

Generate Random Input

The question is what type of input we should produce to test the utility? We found three approaches to generate test data: Generic test data [3], Intelligent test data [3] and fuzz [1]. Generic data is used in generic testing in which we use same test data generator to test all components. Intelligent test data is dependent on the utility being tested. The intelligent data generator produces specific testing string for each utility. Fuzz consist in only generating random data without any specification.

We decided to choose the third technique to generate our inputs, and so to use f*uzz*, a software provided by the previous studies. It produces a continuous string of characters on its standard output. This tool provides interesting options to generate different kind of inputs. It can generate printable and/or non printable characters. Printable characters are those which can be displayed on the screen (for example A,s,2,# etc.) while non printable characters are control characters (for example alt, ctrl etc.). The idea behind giving both printable and non printable characters is that users may give any type of input either deliberately or by mistake. We can decide how many characters we want to generate, and also define the last character of the string, this is a very useful option which allow us to validate a command while testing a program like *bash*.
For example, The following command generate 200 printable characters and write them to output-file:

```
fuzz 200 -p -o output-file
```

The size of the input can have some impact on the result of the test that's why in case of crash we tried to redo the test with some smaller inputs. The basic size of input we used was 1000.

Supply Input to Utilities

We supplied random string of characters generated by *fuzz* to the tested utility. There are two different kinds of utilities :

- Those that we can test directly by sending our input on their standard input. (ex : *grep, wc*)
- Those which require a pty emulator use to send them the input while they are running (ex: *Vim, bash*).

For this purpose we used another tool provided by the previous study: *ptyjig*. It is a software which allows the testers to test interactive utilities. *Ptyjig* basically lies between *fuzz* and target utility. It makes possible to feed a software like a text editor with inputs. While using this tool, the tested software have the impression to receive some characters from the keyboard.

This example show how to test a basic utility on its standard input by sending 200 random characters:

```
fuzz 200 -p | utility
```

The following example demonstrates how we can provide input generated by fuzz to any utility through *ptyjig*:

```
fuzz 200 -p  | ptyjig utility
```

Testing network utilities is slightly different of the other categories. During normal use of the network services the user never interact directly with the server, they go through a program which send the information usually after formatting them according to the correct protocol. The goal of our study is to bypass this programs and send our inputs to the server without following any protocol. To do that we need to use another connector: *portjig*. This tool, very similar to the UNIX tool *netcat*, connect to a specific port of a computer and directly write there all the information we want. So we can use it in the same way as *ptyjig*:

```
fuzz | portjig localhost 22
```

Check the Behavior

After suppling the random inputs to tested utilities, we determine what type of termination these utilities showed. To do that we check the return code of the program. If any utility terminates with desired output or any programmed error message (in case of failure) we categorized that utility as "reliable". Whereas if it hangs or crashes, we classified it as "unreliable".

Test method

In order to correctly test the utilities, we performed the tests several times. We can not rely on only one or two test to determine the reliability of a program. To achieve this goal, we used some scripts. Their purpose is to test

the program a defined number of times. If a failure is discover during one test, the script exits, saving in a file the inputs which make the utility to crash.
A return code different of zero (or/and some other number, depend on the aim of the tested program) indicates a failure. Staying stopped in a middle of a test means that the utility is stuck in an infinite loop.
During testing the big amount of non printable characters that we send to the tested program arrive sometimes on the screen. For example the result of the command *grep* is printed on the screen, whatever it is.
That can lead our graphical terminal to crash. We discover here a non expected failure in the "gnome-terminal".
Solutions are to redirect all the output of the tested program to /dev/null or to use directly a full text terminal which is more basic and stronger.

Categories of utilities

We selected two categories of utilities for our test they are: shell utilities and network utilities

Shell utilities

We can directly test this kind of utility with the fuzz program without any interface. In that case, it is relatively simple to use. For more complicated software like a shell, we have to use the utility ptyjig which create a virtual terminal as interface between fuzz and its target. The utilities we selected for this category are:

- *grep, uniq* and *wc* because they are probably some of the most often used
- *Indent* because it was crashing in the old studies
- b*ash* for testing how an interactive program react to these random inputs

Network utilities

In this category we need to use the tool provided in the previous study : *portjig*.
We decided to test following services:

- *ftpd as* one of the most basic unix services. We didn't want to test an evolved ftp server such as *proftpd*, but a real basic one. This service was run by the internet superserver *inetd*. We also choose it because it already had some weakness in the past.
- *Apache* as the most common web server on the internet it should be able to react in a smart way to all the input we send it, and it shouldn't have such security hole.
- *Postfix* as a very powerful mail server should also be very strong against this kind of aggression.
- sshd as a wildly used deamon providing a remote shell We tought interesting to check its behavior in such situation.

RESULT AND ANALYSIS

Shell based utilities

We will show here the result of our test in table 1.
There are two columns for the result as for each utility we test its resistance both to printable and non printable characters.

Commands		
	Input Characters	
Utility	**Printable**	**Non Printable**
grep 2.5.1	Success	Success
wc 5.2.1	Success	Success
uniq 5.2.1	Success	Success
bash 3.00.16	Success	Crash
indent 2.2.9	Success	Success

Table 1: Test results of shell utilities

As we can see on the table above all the basic utilities tested pass the tests, even *indent* which was failing in the last studies.
However, it seems that the shell bash have some problems with the non printable inputs as it often crashed during theses test. This can be surprising as it is one of the most commonly used shell of the non professional users. However it handled the printable inputs and as it is only suppose to receive characters from the keyboard, that should not be a problem in a classical situation. We also tried to give it some smaller input or to change the speed at which the characters are provided, but that didn't change the result of the test.
To check if the bugs are corrected, we decided to select one utility crashing in the last study and to reproduce the bug with the old version and then to submit the new version to the same test. We did that on the utility *indent*, this is a small utility used to indent the source code correctly (it is one of the tools used by the vim editor to format the code in the right way). We saw that the bug was corrected after the study, as this new version accept the wrong input without any problems.

Network services

We submit our selected network utilities to the same test as the shell based. Table 2 shows the results of tests of network based utilities :

Network			
	Input Characters		**Description**
Utility	**Printable**	**Non Printable**	
ftpd 0.17-20	hang	hung	Service terminated by 'inetd' because of infinite loop

Network			
	Input Characters		Description
Utility	Printable	Non Printable	
apache 1.3.33	Success	Success	Connection close
postfix 2.2.4	Success	Success	Connection close
Openssh 4.1	Success	Success	Connection close

Table 2: Test results of network utilities

As we can see apache postfix and ssh always react by simply closing the connection, without being disturbed. They simply notice the wrong inputs in their respective logs and wait for the next connection. That policy is really good and allow them to be protected from any error which can be caused by any random inputs.

However the more basic service ftp was stopped by its handling process inetd because it was stuck in an infinite loop. We could see a difference with the other servers because in contrast of them ftpd did not close the connection when it received our wrong inputs. It is visible in the logs, where all the inputs were wrote, sometimes separated by the errors messages of ftpd. The non application of this secure policy consisting of closing the connection led it to theses instabilities.

By the way it is interesting to make this observation because this service was crashing in the study of 1990 and the problem was corrected in the 1995 study.

We tried to find the string which was able to crash the daemon and we discovered that this string used a second time, did not affect *ftpd*. Then we tried to change the speed at which we send the input so the server and we discovered that when the inputs are sent slower *ftpd* does not hang and behave normally. The problem with *ftpd* is so mainly a problem related to the speed of the input more than the kind of inputs. This kind of error probably come from the size of a buffer.

General result

According to our test we could see that some of the utilities such as *indent* were improved since the last study and some other like *ftpd* went in the wrong way.

Not all the program we tested were tested in the previous studies, so we can not make comparison along their evolution for all of them, but we thought interesting to test the very commonly used one. Also we could see that surprisingly it was easier to make them crash than we expected.

We also found out that the gnome-terminal had some problems to handle out non printable inputs. That show us that every program have its vulnerabilities and that it is more common than we could think.

However the application of secure principle while conceiving program can really help to improve the stability of the programs as we saw in the case of *ftpd* which does not close the connection when he receives improper inputs.

Comparison between different categories

The main difference between the network and the shell based utilities come from the protection method used by the network applications. Their ability to close connection to anybody sending wrong inputs help them a lot for their self-protection. Whereas the shell based utility always process their inputs whatever they are and do not have the possibility to block any stream of input arriving to them.

Comparison between different versions

utilities		
	Tested in :	
Utility	1995	2006
grep 2.5.1	Success	Success
wc 5.2.1	Success	Success
uniq 5.2.1	Success	Success
bash 3.00.16	X	Crash
indent 2.2.9	Crash	Success
ftpd 0.17-20	Success	Hung
apache 1.3.33	Success	Success
postfix 2.2.4	Success	Success
Openssh 4.1	Success	Success

Table 3 : comparison of versions
(X = not tested)

In table 3 we can see that on one hand *indent* was improved since the last studies and on the other hand *ftpd* got some more problems since the last study. The other tools tested did not have any problems before and are still non sensitive to the random inputs.

CONCLUSIONS

After performing the various tests on a number of common interactive UNIX utilities, we conclude that there is a significant improvement over the reliability of these utilities. Most of new versions of the utilities removed their old vulnerabilities.

But still there exist utilities that crash or hang in course of getting different kind of inputs. The reason for this are:

first: most of these utilities are not developed in context

of security or reliability and do not make use of some basic policy, so developers do not emphasise on reliability part.

Second: some flaws occur due to programming logic

finally: software testing does not performed in suitable way.

We also saw that the application of basic security principles are really important to improve the overall reliability of the program.

We can see that these basic tests already performed in the previous study are still useful even some years after.

There are many UNIX utilities and we selected few of them. We can not claim that our work provide test results of reliability of all utilities. However, our work can be extended to test more usable utilities.

REFERENCES

[1]. Barton P. Miller, Lars Fredriksen, Bryan, “An empirical study of the reliability of UNIX utilities”, *Communications of the ACM*, Pages: 32 - 44, year of publication 1990.

[2]. Barton P. Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, Jeff Steidl, “fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services”, Department of Computer Science, University of Wisconsin, 1995.

[3]. Matthew Schmid & Frank Hill, "Data Generation Techniques for Automated Software Robustness Testing", Reliable Software Technologies Corporation 21515 Ridgetop Circle #250, Sterling, VA 20166.