Using the Linux Operating System to Illustrate Message Digest Concepts and Vulnerabilities

Renat Sultanov
Chemistry Department
University of Nevada, Las Vegas,
Las Vegas, NV 89154
sultano2@unlv.nevada.edu

Dennis Guster and Paul Safonov
BCIS Department
St. Cloud State University,
St. Cloud, MN 56301
dcguster@stcloudstate.edu and safonov@stcloudstate.edu

Abstract

There are many ways to help insure file integrity, but one of the most common is the application of a message digest algorithm. A message digest is a fixed string number which represents that file uniquely. It is widely believed that Information Technology students typically learn most effectively when given meaningful hands on exercises. Therefore, examples using the Linux operating system were devised. Hence, the purpose of this paper was to use the commands with in the Linux operating system to illustrate message digest concepts and vulnerabilities. Specifically, the paper addressed using date, time and file size as integrity checks and discussed their limitation in light of the capabilities of the touch command. The parity check algorithm was applied to files to illustrate how a simple algorithm works. Sum, cksum and md5sum were applied to a series of files to illustrate differences in robustness and efficiency. An md5sum vulnerability identified by Kaminsky was coded and tested. Other digests were applied to the same file series to illustrate that the vulnerability is an anomaly unique to md5sum and not other digests. The implications of recording more than one digest for a given file were discussed.

1 Introduction

Unauthorized attacks on information systems continue to grow. In fact, the Computer Crime Research Organization reports that hacker attacks grew 37% in the first quarter of 2004 (Keefe, 2004). One method of attack is aimed at modifying the contents of legitimate system and data files. There are many scenarios, but two common schemes involve either modifying the data to disrupt operations or placing executable code in the file in hopes of providing the hacker with a back door entry into the system. Spurgeon and Schaefer (2004) in System Administrator Magazine state that verifying the integrity of files is an important systems administration task.

There are many ways to help insure file integrity, but one of the most common is the application of a message digest algorithm. A message digest is a fixed string number which represents that file uniquely. If the content of that file change, then the message digest will change as well. The digest commands (such as cksum, md5sum, sha1) calculate file digests, which are sometimes referred to as ``fingerprints'' or ``hashes'', of the contents of that file. Structurally, the digest is usually a small fixed string. The actual size of the digest depends of the algorithm used. Furthermore, digest formulas are one-way functions, making it very difficult to find a file that matches any given digest by trial and error. There is a wide variety of algorithms available which differ in robustness and efficiency.

2 A Pedagogical Approach

Information Technology students typically learn most effectively when given meaningful hands on exercises. The Linux operating system because of its openness and flexibility is well suited to support this goal. Therefore, the purpose of this paper was to use the commands with in the Linux operating system to illustrate message digest concepts and vulnerabilities. Specifically, the paper addressed using date, time and files size as integrity checks and discusses their limitation in light of the capabilities of the touch command. The parity check algorithm was applied to file to illustrate how a simple algorithms work. Sum, cksum and md5sum were applied to a series of files to illustrate differences in robustness and efficiency. An md5sum vulnerability identified by

Kaminsky which generated much discussion on the net (Kerner,2004 and Mearian, 2005) was coded and tested. Other digests will be applied to the same file series to illustrate that the vulnerability is an anomaly unique to md5sum and not other digests. The implications of recording more than one digest for a given file will be discussed.

3 Linux Examples

For any given file that is created on a file system it is possible that that file could be tampered with by hackers. The severity of the attack could vary from deleting or modifying the content of that file to placing an executable in the file that may provide a backdoor entry or launch a virus. It is possible to protect the integrity of any given file by employing an algorithm that will provide a digital signature (message digest) of that file's contents. For this process to be effective a library of file digital signatures must be maintained so that the file can be compared to its historical "correct" digital signature before it is read in the future. The Unix operating system provides a number of commands that make it easy to illustrate this process.

3.1 Date, Time and Files Size as Integrity Checks

Using date, time and file size can provide a very basic integrity check, however it is very limited in effectiveness because the date, time and file size value can be easily tampered with.

First we create a file called fileintg:

```
bcis501@forum:~$ cat > fileintg
dog
```

This file was created by redirecting the keyboard output to a file in the current directory called fileintg. The file's listing in the current directory appears below. The creation date and size can provide a very basic integrity check

```
bcis501@forum:~$ ls -al fileintg
-rw-r--r--    1 bcis501  user          4 Dec  5 11:06 fileintg
```

From this output we can determine the file contains 4 bytes. A hex dump show that actually the file contains the ASCII string "dog" and a hex new line symbol :0a".

```
bcis501@forum:~$ xxd fileintg
0000000: 646f 670a                        dog.
```

The size and the creation date can be easily logged to a history file:

First, the output is placed in a temporary work file named lscut.
bcis501@forum:~$ ls -al fileintg > lscut

Second, the contents are saved to a file called intglog, which would be viewed as the system master integrity file. The output is also sent to the screen through use of the tee command.

bcis501@forum:~$ cut -b 40-65 lscut | tee intglog
 4 Dec 5 11:06 fileintg

A file created at a later date could easily be overlaid under the original file name. Below a different version of fileintg is created on January 1. The date can be easily changed with the touch command.

bcis501@forum:~$ ls -al fileintg
-rw-r--r-- 1 bcis501 user 4 Jan 1 2005 fileintg

bcis501@forum:~$ touch -m -t 200412051106 fileintg
bcis501@forum:~$ ls -al fileintg
-rw-r--r-- 1 bcis501 user 4 Dec 5 11:06 fileintg

Note: The date, time and size now match exactly, however the contents are different.

bcis501@forum:~$ cat fileintg
cat

3.2 Parity Checking

There are more sophisticated methods of determining if a file has been tampered with that rely on the bit pattern within the file.

Given the original fileintg file.

bcis501@forum:~$ xxd fileintg
0000000: 646f 670a dog.

The 4 characters could be converted to binary from the hex string above and it would yield:

0110 0100 0110 1111 0110 0111 0000 1010

Parity checking can be applied, in which the number of 1's in the string are determined, in this case there are 16, which could be viewed a very simple quantitative integrity check for this file. What if the file contained the string coi? The binary appears below. It still

contains 16 one's but in a different pattern. So therefore, it doesn't provide the needed sophistication.

0110 0011 0110 1111 0110 1001 0000 1010

3.3 More Sophisticated Formulas

Linux provides three different formulas of varying sophistication: sum, cksum and md5sum all three more robust that parity checking.

The sum command produces a 5 digit (checksum) decimal signature for fileintg, so in terms of robustness the probability of guessing the signature is 10^5. (note the 1 below is the number of blocks contained in the file).

```
bcis501@forum:~$ sum fileintg
49253     1
```

The cksum command produces a 10 digit (CRC) decimal signature for fileintg, so in terms of robustness the probability of guessing the signature is 10^10. (note the 4 is the number of bytes contained in the file).

```
bcis501@forum:~$ cksum fileintg
1975358332 4 fileintg
```

The md5sum command produces a 32 digit (md5sum) hexadecimal signature for fileintg, so in terms of robustness the probability of guessing the signature is 16^32.

```
bcis501@forum:~$ md5sum fileintg
362842c5bb3847ec3fbdecb7a84a8692  fileintg
```

So therefore in terms of robustness it is clear that the md5sum is the most sophisticated, however in terms overhead from the added complexity how do the three algorithms scale? The time command can be placed in the command string to provide an idea of how long it takes each to execute. On small files like fileintg there is little difference in the time to calculate. In fact the elapsed time (real) in each case is .003ms (see the results below).

```
bcis501@forum:~$ time sum fileintg
49253     1

real    0m0.003s
user    0m0.010s
sys     0m0.000s
```

```
bcis501@forum:~$ time cksum fileintg
1975358332 4 fileintg

real    0m0.003s
user    0m0.000s
sys     0m0.010s

bcis501@forum:~$ time md5sum fileintg
362842c5bb3847ec3fbdecb7a84a8692  fileintg

real    0m0.003s
user    0m0.000s
sys     0m0.000s
```

On large files, well in excess of 1 million bytes there is still little difference in performance. See the termcap example below.

```
bcis501@forum:~$ ls -al t*
-rw-r--r--    1 bcis501  user      1391457 Sep 15 11:25 termcap

bcis501@forum:~$ time sum termcap
62097  1359

real    0m0.019s
user    0m0.010s
sys     0m0.000s

bcis501@forum:~$ time cksum termcap
3091109216 1391457 termcap

real    0m0.014s
user    0m0.010s
sys     0m0.000s

bcis501@forum:~$ time md5sum termcap
4d562dfa3c6cc9d55c5c79bbc31f4e97  termcap

real    0m0.015s
user    0m0.010s
sys     0m0.000s
```

3.4 Automating the Verification Process

The process of logging and checking the logged digital signature can easily be automated thru an OS script file.

```
bcis501@forum:~$ cat dsfile
echo create the file
pico $1
echo save its digital sum
md5sum $1 > md5his
echo next time it is accessed
echo calculate its md5sum and store for comparison
(sleep 10; md5sum $1 > md5now)
cmp md5his md5now
if [ $? == 0 ]
then
        cat $1
else
        echo ds do not match
fi
echo $?
echo end of script
```

Here is the execution of that script:

```
bcis501@forum:~$ ./dsfile rer
create the file
  UW PICO(tm) 4.6                File: rer

this is a sample file

                        [ Wrote 1 line ]

save its digital sum
next time it is accessed
calculate its md5sum and store for comparison
this is a sample file
0
end of script
```

3.5 Vulnerabilities in md5sum

Even as strong as md5sum appears it is still vulnerable: An example published by Dan Kaminsky (2004), entitiled: MD5 To Be Considered Harmful Someday appears below.

```
$vec1 = h2b("
d1 31 dd 02 c5 e6 ee c4 69 3d 9a 06 98 af f9 5c
2f ca b5 87 12 46 7e ab 40 04 58 3e b8 fb 7f 89
55 ad 34 06 09 f4 b3 02 83 e4 88 83 25 71 41 5a
08 51 25 e8 f7 cd c9 9f d9 1d bd f2 80 37 3c 5b
d8 82 3e 31 56 34 8f 5b ae 6d ac d4 36 c9 19 c6
dd 53 e2 b4 87 da 03 fd 02 39 63 06 d2 48 cd a0
e9 9f 33 42 0f 57 7e e8 ce 54 b6 70 80 a8 0d 1e
c6 98 21 bc b6 a8 83 93 96 f9 65 2b 6f f7 2a 70
");

$vec2 = h2b("
d1 31 dd 02 c5 e6 ee c4 69 3d 9a 06 98 af f9 5c
2f ca b5 07 12 46 7e ab 40 04 58 3e b8 fb 7f 89
55 ad 34 06 09 f4 b3 02 83 e4 88 83 25 f1 41 5a
08 51 25 e8 f7 cd c9 9f d9 1d bd 72 80 37 3c 5b
d8 82 3e 31 56 34 8f 5b ae 6d ac d4 36 c9 19 c6
dd 53 e2 34 87 da 03 fd 02 39 63 06 d2 48 cd a0
e9 9f 33 42 0f 57 7e e8 ce 54 b6 70 80 28 0d 1e
c6 98 21 bc b6 a8 83 93 96 f9 65 ab 6f f7 2a 70
");
```

Note that the bold characters in the dump have been modified in vec2 from the contents of the original vec1.

To test the md5sum vulnerability we first remove the labels and spaces from the out put above:

```
bcis501@forum:~$ cat vec1b
d131dd02c5e6eec4693d9a0698aff95c
2fcab58712467eab4004583eb8fb7f89
55ad340609f4b30283e488832571415a
085125e8f7cdc99fd91dbdf280373c5b
d8823e3156348f5bae6dacd436c919c6
dd53e2b487da03fd02396306d248cda0
e99f33420f577ee8ce54b67080a80d1e
c69821bcb6a8839396f9652b6ff72a70

bcis501@forum:~$ cat vec2b
d131dd02c5e6eec4693d9a0698aff95c
2fcab50712467eab4004583eb8fb7f89
55ad340609f4b30283e4888325f1415a
085125e8f7cdc99fd91dbd7280373c5b
d8823e3156348f5bae6dacd436c919c6
dd53e23487da03fd02396306d248cda0
e99f33420f577ee8ce54b67080280d1e
c69821bcb6a8839396f965ab6ff72a70
```

Then convert the hex to characters:

```
bcis501@forum:~$ xxd -l 120 -ps -r -c 20 vec1b > vec1c
bcis501@forum:~$ xxd -l 120 -ps -r -c 20 vec2b > vec2c
```

vec1&2c now appear as 128byte non-ASCII files:

```
bcis501@forum:~$ ls -al v*
-rw-r--r--    1 bcis501  user        385 Dec 14 16:16 vec1
-rw-r--r--    1 bcis501  user        265 Dec 15 17:44 vec1b
-rw-r--r--    1 bcis501  user        128 Dec 15 17:46 vec1c
-rw-r--r--    1 bcis501  user        385 Dec 14 16:17 vec2
-rw-r--r--    1 bcis501  user        265 Dec 15 18:01 vec2b
-rw-r--r--    1 bcis501  user        128 Dec 15 18:01 vec2c
```

Md5sum each one:

```
bcis501@forum:~$ md5sum vec1c
79054025255fb1a26e4bc422aef54eb4  vec1c

bcis501@forum:~$ md5sum vec2c
79054025255fb1a26e4bc422aef54eb4  vec2c
```

Note the hashes are identical, but the diff command catches the differences on the character level.

```
bcis501@forum:~$ diff -d vec1c vec2c
1c1
< Ñ1ÝÅæîÄi=¯ù\ÊµF~«@X>¸ûU-4
ô³ä%qAQ%è÷ÍÉÙ½ò7<[Ø>1V4[®m¬Ô6ÉÆÝSâ´Úý9cÒHÍ é3BW~Æ!¼¶¨ùe+o÷*p
\ No newline at end of file
---
> Ñ1ÝÅæîÄi=¯ù\ÊµF~«@X>¸ûU-4
ô³ä%ñAQ%è÷ÍÉÙ½r7<[Ø>1V4[®m¬Ô6ÉÆÝSâ4Úý9cÒHÍ é3BW~Æ!¼¶¨ùe«o÷*p
\ No newline at end of file
```

It is interesting to note that sum and cksum both detect the difference. Perhaps using multiple message digests would provide better security, but with more overhead. Also, the odds of finding an anomaly that would defeat two or more algorithms at the same time would appear to be quite small.

```
bcis501@forum:~$ sum vec1c
11671     1
bcis501@forum:~$ sum vec2c
07575     1

bcis501@forum:~$ cksum vec1c
4283120310 128 vec1c
bcis501@forum:~$ cksum vec2c
2258160912 128 vec2c
```

4 Conclusions

Although lecture and discussion of concepts are a well establish methods in computer education, they tend for the most part to be passive in nature. To truly understand may of the complex concepts within computing students need an opportunity for experimentation in the computing environment. The Linux operating system because of its openness and flexibility provides an excellent platform to facilitate that experimentation. The body of the paper herein was designed to provide an example of how that experimentation could be applied to gain a better understanding of the file integrity problem. The presentation started out simple, but with a little research on the internet was able to introduce a fairly complex anomaly involving the Kaminsky problem.

Students typically prefer the hands-on approach when compared to a lecture only alternative. Instructors following the hands-on approach often get comments like "I really truly didn't understand it until I ran the experiment myself!" Industry advisory groups often favor the approach as well because students graduating from programs that feature the hands-on approach are often better prepared for the world of work.

References

Kaminsky, D. (2004). "MD5 To Be Considered Harmful Someday", www.doxpara.com/**md5**_someday.pdf .

Keefe, B. (2004). "Computer Crime Research Organization News", http://www.crime-research.org/news/2003/04/Mess0902.html.

Kerner, S. (2004). "MD5 Flaw Threatens File Integrity", http://www.esecurityplanet.com/patches/article.php/3446071

Mearian, L. (2005)."Content addressed storage systems may be at risk", http://www.computerworld.com/hardwaretopics/storage/story/0,10801,99331,00.html.

Spurgeon, J. and Schaefer, E. (2004). "Managing Enterprise Alerts", System Admin, 13(4).