Application Self-Validation Using MD5

by

Todd M. Mizenko
Dot4, Inc.

June 20, 2005

© Copyright 2005 Dot4, Inc.

Dot4, Inc. • 6080 Jericho Turnpike, Suite 211 • Commack, New York 11725
www.dot4.com • 800-834-1951

©2005 Dot4, Inc.

Table of Contents

1 ABSTRACT 3

1.1 DEFINING TERMS 3

2 DESIGN 4

2.1 PARSING AN ELF FILE 6
2.2 MD5 ENGINE 9
2.3 EMBEDDING APPLICATION 10
2.4 TEST APPLICATION 10
2.5 PUTTING IT ALL TOGETHER 11
2.6 VARIATIONS 13
2.6.1 Split Checksums 16
2.6.2 One-shot Threaded 16
2.6.3 Continuous Threaded 16
2.7 ENHANCEMENTS 17

3 CONCLUSION 18

4 APPENDIX – SOURCE CODE 19

4.1 MD5 ENGINE 19
4.1.1 md5_.h 19
4.1.2 md5.c 20
4.2 COMMON HEADER – MAGIC.H 30
4.3 EMBEDDING APPLICATION – EMBED.C 31
4.4 TEST APPLICATION – TEST.C 34
4.4.1 One-shot Threaded – test1.c 36

5 REFERENCES 38

Figures

FIGURE 1 – SMALL FILE MD5 COMPUTATION TIME VS. BUFFER SIZE 15
FIGURE 2 – LARGE FILE MD5 COMPUTATION TIME VS. BUFFER SIZE 15

Tables

TABLE 1 - BUFFER SIZE VS. TIME 14

©2005 Dot4, Inc.

1 Abstract

Certain Unix executables may benefit from a system by which the executable can verify its own integrity on startup. This verification may be used for multiple purposes, including helping to prevent unauthorized modification (for example, to defeat copy protection mechanisms) and to prevent a bad copy of the program from running (for example, due to an erroneous copy from one media to another). The latter was the initial motivation for this research.

The network being used for development on a new project was being reported to have data loss problems. Running tests using ftp (file transfer protocol) to move files around the network and then calculating MD5 sums on each resulted in a differing checksums. Since executable code was routinely moved around the network in a similar manner, the author decided during the design phase of this new project to explore ways to have executable code self-validate on startup. This white paper is result of that exploration.

The reader is assumed to be both knowledgeable and proficient in C programming and Unix. Familiarity with Posix threads is also helpful.

1.1 Defining Terms

- **MD5 sum**
  From Wikipedia[1], the free encyclopedia: In cryptography, MD5 (Message-Digest algorithm 5) is a widely used cryptographic hash function with a 128-bit hash value. As an Internet standard (RFC 1321[2]), MD5 has been employed in a wide variety of security applications, and is also commonly used to check the integrity of files. May also be referred to in this document as **MD5**, **sum**, **checksum**, or **MD5 checksum**.[3]

- **ELF**
  Executable and Linking Format[4]. The standard file format for many Unix-variant executables. All of the executables referenced in this paper are ELF format.

©2005 Dot4, Inc.

2 Design

From the outset two things were known: first, because of its strength and ubiquitous nature, the validation mechanism of choice was MD5 over standard CRC32, etc. Second, to be effective, space within the program for the MD5 must be created at compile time. That way, at run time the program would theoretically have access to the MD5 sum.

With this in mind, an initial experiment was run on Linux (RedHat version 7.1) to see how different types of global data were placed within a test executable. Since an MD5 sum is 128 bits in length a 16-byte character array (16 bytes x 8 bits/byte = 128 bits) was used as the placeholder for the MD5 sum. Three different 16-bytes arrays were declared at the top of a simple C source file (`test.c`), which is show below.

```
#include <stdio.h>

const char var1[16] = "fahoogiboots123";
char var2[16] = "plastidip123456";
char var3[16];

int
main(void)
{
        printf("%s %s\n", var1, var2);
        var3[0] = 'a';
}
```

The source was then compiled and dumped using **od** (`-c` option for dumping characters) to examine the contents of the executable. The string contents were located at location 2370 octal:

```
0002360 003  \0  \0  \0 001  \0 002  \0   f   a   h   o   o   g   i   b
0002400   o   o   t   s   1   2   3  \0   %   s       %   s  \n  \0  \0
0002420  \0  \0  \0  \0  \0  \0  \0  \0   @ 225 004  \b  \0  \0  \0  \0
0002440   p   l   a   s   t   i   d   i   p   1   2   3   4   5   6  \0
0002460  \0  \0  \0  \0   ?   ?   ?   ?  \0  \0  \0  \0   ?   ?   ?   ?
```

Further examination showed the location of the string variables themselves, starting at location 32270 octal:

```
0032260  \0   _   f   p   _   h   w  \0   v   a   r   1  \0   _   i   n
0032300   i   t  \0   _   _   d   e   r   e   g   i   s   t   e   r   _
0032320   f   r   a   m   e   _   i   n   f   o   @   @   G   L   I   B
0032340   C   _   2   .   0  \0   _   s   t   a   r   t  \0   _   _   b
0032360   s   s   _   s   t   a   r   t  \0   m   a   i   n  \0   _   _
0032400   l   i   b   c   _   s   t   a   r   t   _   m   a   i   n   @
0032420   @   G   L   I   B   C   _   2   .   0  \0   v   a   r   2  \0
0032440   v   a   r   3  \0   d   a   t   a   _   s   t   a   r   t  \0
0032460   p   r   i   n   t   f   @   @   G   L   I   B   C   _   2   .
```

Next, using **nm** it was noted which section each variable was in (addresses are in hex):

```
080484f8 R var1
08049520 D var2
08049620 B var3
```

So var1 was placed in the **.rodata** (‘R’) section, `var2` in the **.data** (‘D’) section and `var3` in the **.bss** (‘B’) section. The locations for `var2` and `var3` were confirmed by noting that they lie shortly after the start address of their respective sections (this version of **nm** did not display a start address for the **.rodata** section):

```
08049608 A __bss_start
         w __cxa_finalize@@GLIBC_2.1.3
08049510 D __data_start
```

This also jibes with what was learned from concurrent background research into the ELF format[5]. Another way to look at these variables is by using the utility readelf. Dumping the ELF headers showed **.rodata** in section 14 and **.data** in section 15. Dumping these sections shows the initialized data:

```
% readelf -x14 test

Hex dump of section '.rodata':
  0x080484f0 6269676f 6f686166 00020001 00000003 ........fahoogib
  0x08048500   000a73 25207325 00333231 73746f6f oots123.%s %s..

% readelf -x15 test

Hex dump of section '.data':
  0x08049510 00000000 08049540 00000000 00000000 ........@.......
  0x08049520 00363534 33323170 69646974 73616c70 plastidip123456.
```

Because an uninitialized global variable (`var3`, **.bss** section) allocates no space it would not be possible to locate or embed a checksum using one. Therefore, either an initialized global (`var2`, **.data**) or constant initialized variable (`var1`, **.rodata**) became the candidates for storing the checksum. In the end, it was decided to use `var1` in **.rodata** simply because using a variable declared as `const` will prevent the executable from modifying its value.

At this point the problem could be broken up into two major components and two minor components. The major components were:

- Parsing an ELF file to the location containing the checksum
- Creating an MD5 engine to compute and embed the checksum

The minor components were:

- Creating a post-compilation utility that embeds the checksum into the linked test executable
- Creating a test program that validates itself using the major components

All components are discussed below.

2.1 Parsing An ELF File

To understand how to parse an ELF executable with global data placed in the **.rodata** section, an understanding of the ELF layout is essential[6]. The basic layout is represented in the table below.

ELF Header
Program Header (optional)
Section 1
…
Section *n*
…
…
Section Header Table

Every ELF file starts with an ELF Header that allows it to be easily identified as such by checking the first four bytes. They should be 0x7F, followed by the characters 'E', 'L', and 'F'. By opening the file and using `lseek()` to move the file descriptor to location 0, the ELF Header can be loaded into a variable of type `Elf32_Ehdr`. This data type contains other important fields that allow the user to verify the object file type (`e_type`), ELF version (`e_version`), and machine class (`e_ident[EI_CLASS]`). When performing ELF operations all of these fields are validated before proceeding. The object file type should be executables (`ET_EXEC`), the version should be 1, and the machine class should be 32-bit (`ELFCLASS32`). To aid in understanding, code snippets are provided (see Section 4.1.2 for a complete listing).

```
if(lseek(fd, 0, SEEK_SET) != 0) {
       ...

if(read(fd, &elf_hdr, sizeof(Elf32_Ehdr)) != sizeof(Elf32_Ehdr)) {
       ...

if(strncmp((const char *)elf_hdr.e_ident, ELFMAG, SELFMAG) != 0) {
       ...

if((elf_hdr.e_type != ET_EXEC) || (!(elf_hdr.e_version)) ||
   (elf_hdr.e_ident[EI_CLASS] != ELFCLASS32)) {
       ...
```

The ELF Header structure (type `Elf32_Ehdr`) contains several additional useful fields used in locating the global data placed in **.rodata**:

`e_shoff`
Holds the Section Header Table's file offset (in bytes). This table is used first to locate the string table section and second to locate the .rodata section.

`e_shnum`
Holds the number of entries in the Section Header Table. This value is used to determine how big the Section Header Table is as well as how many sections to iterate over while looking for **.rodata**.

`e_shstrndx`
Holds the Section Header Table index of the string table. This index allows for direct access of the string table.

The ELF Section Header structure (type `Elf32_Shdr`) describes each section present in the Section Header Table. Several fields from this structure are also useful for the task at hand:

`sh_size`
Holds the section's size in bytes. This field is needed when allocation space for storing the string table as well as **.rodata**.

`sh_offset`
Holds the section's offset from the beginning of the file in bytes. This field is needed when file seeking in order to store the string table as well as **.rodata**.

`sh_name`
Specifies the section's name as an offset into the Section Header string table section. This field is used to locate the section index for **.rodata**.

The task then becomes putting these data structures to work in actually locating the embedded string. At this point, the ELF file basics have been validated. Since all the required offsets, indices, etc. live in the Section Header Table, the first things to do are: compute the table size, allocate memory for the table, seek to the table in the file, and read in the table. The table size is simply the number of sections (represented by the `e_shnum` member of `Elf_32Ehdr`) multiplied by the size of a section header (`sizeof(Elf32_Shdr)`). The allocation and read steps are standard C-language operations that the reader is assumed to be familiar with. The seek operation is as well with the note that the seek location is the file offset to the Section Header table or `e_shoff`. Once the Section Header Table is loaded, the string table needs to be loaded next as each section has its name specified there.

```
size = elf_hdr.e_shnum * sizeof(Elf32_Shdr);
if((sect_hdr = (Elf32_Shdr *)malloc(size)) == NULL) {
       ...

if(lseek(fd, elf_hdr.e_shoff, SEEK_SET) != elf_hdr.e_shoff) {
       ...

if(read(fd, sect_hdr, size) != size) {
       ...
```

Since the ELF header directly provides the index for the string table, loading it into memory is fairly straightforward. The size (`sh_size`) and file offset (`sh_offset`) of the table are specified by indexing into the Section Header Table with `e_shstrndx`. A similar process as above is followed for loading this table.

```
str_tab_size = sect_hdr[elf_hdr.e_shstrndx].sh_size;
if((string_table = calloc(str_tab_size, 1)) == NULL) {
       ...

if(lseek(fd, sect_hdr[elf_hdr.e_shstrndx].sh_offset, SEEK_SET) !=
   sect_hdr[elf_hdr.e_shstrndx].sh_offset) {
       ...
```

```
if(read(fd, string_table, str_tab_size) != str_tab_size) {
      ...
```

As presented earlier, the `sh_name` member of each section header specifies the section's name as an offset into the string table. Therefore, since the string table is now loaded it is simple to iterate over the number of sections and locate **.rodata** in the string table.

```
for(idx=0; idx < elf_hdr.e_shnum; idx++) {
      if(strncmp((string_table + sect_hdr[idx].sh_name),
         ".rodata", 7)==0) {
```

Once the section name is located in the string table, the section index to **.rodata** then becomes known directly. Therefore, the next step is to read in this section.

```
if(lseek(fd, sect_hdr[idx].sh_offset, SEEK_SET) !=
   sect_hdr[idx].sh_offset) {
       ...

if((rodata = malloc(sect_hdr[idx].sh_size)) == NULL) {
       ...

if(read(fd, rodata, sect_hdr[idx].sh_size) !=
   sect_hdr[idx].sh_size) {
       ...
```

The only things left to do at this point are search through **.rodata** looking for the global magic string and once that's found, computing the file offset to the global magic string location. The global magic string is located by performing repeated string compares over the section, moving the section pointer one byte at a time until the compare passes. When it does, the data has been located! The offset to the string therefore is simply the **.rodata** section offset plus the number of bytes incremented into the file to get the string compare to pass.

```
magic_ptr = rodata;

while(magic_ptr < (rodata + sect_hdr[idx].sh_size)) {
      if(strncmp(magic_ptr, magic_str, magic_size) == 0) {
            offset = magic_ptr - rodata;
            checksum_offset=sect_hdr[idx].sh_offset+offset;
            break;
      }
      magic_ptr++;
}
```

With the data located (the balance of the source is cleanup and bullet-proofing), the next major task is creating the MD5 engine.

2.2 MD5 Engine

The MD5 engine is a C source module that contains all of the functions necessary for MD5 calculation and manipulation. Although it could, this engine does not implement the MD5 algorithm directly but rather, relies on the open source MD5 library functions that ship standard with most versions of Unix. These functions, along with a brief description, are:

`MD5Init()` - initializes an MD5 context structure for performing the checksum.
`MD5Update()` - updates the MD5 sum based on the passed data and size.
`MD5Final()` - completes the checksum process and returns the checksum.

The MD5 engine was relatively simple to design. The core functionality from which the engine is based on is the routine `MD5_checksum_calc()` which uses the open source MD5 library functions above to calculate a standard MD5 checksum. Knowledge gained here was the basis for the rest of the engine, save `MD5_csum_offset()` (see previous section). The engine contains several functions, all of which assume that the checksum is 16 bytes (128 bits) in length. These functions, along with a brief description, are:

`MD5_csum_offset_get()`
Function outlined in the prior section that calculates the offset into the given file based on the specified checksum string.

`MD5_embedded_csum_calc()`
Function that computes an MD5 for the specified file, skipping the region specified by a checksum offset. This area is used to embed a checksum after compilation and linking. It is essentially `MD5_checksum_calc()` (shown next) broken into two pieces.

`MD5_checksum_calc()`
Function that computes a traditional (contiguous) MD5 sum over the specified file.

`MD5_checksum_embed()`
Function that embeds the specified checksum at the specified offset.

`MD5_checksum_extract()`
Function to extract a previously embedded checksum at the specified offset into the specified buffer.

`MD5_checksum_print()`
Convenience routine for dumping a checksum to standard out. Output format matches that of the standard Solaris **`md5`** utility.

`MD5_buffer_size_set()`
Accessor function used to tune MD5 computation buffer size.

`MD5_buffer_size_get()`
Accessor function used to query the current MD5 computation buffer size.

For diagnostic purposes, most of these functions return –1 on error and print a string describing the error. Feel free to modify them for your specific needs (logging, error checking, etc.). Please see Section 4.1 for a full source listing.

2.3 Embedding Application

The job of the embedding application (`embed.c`) is to post-process the executable in question by calculating and embedding an MD5 checksum into the executable after compiling and linking. The resulting application can then run a similar series of checks on startup (or in a variety of other ways) to perform self-validation. The embedding application and test application/executable both rely on the same string for locating the checksum in the **.rodata** section. The former uses the string to locate the checksum location in the latter, whereas the latter uses the string to stake out space for the checksum in the executable and so that the former can find the checksum.

The basic flow is as follows:

1. Open the executable that is to have the checksum embedded (passed in as `argv[1]`) in read/write, binary mode.
2. Retrieve the checksum offset into the file based on the common magic string.
3. Calculate the checksum.
4. Embed the checksum.
5. Verify the embedding process.

Please see Section 4.2 for a full listing of source code.

2.4 Test Application

With the MD5 engine and embedding application in hand, the test application (`test.c`) is a simple program used to apply the technique of embedding and verifying a checksum. It performs a very small number of operations to accomplish its task, which is self-validation.

The basic flow is as follows:

1. Include the common magic string definition from `magic.h`.
2. Create a global variable of size `D_MAGIC_SIZE` (17, one extra character for trailing NULL) to reserve space for the checksum in the **.rodata** section of the executable and sets it equal to the common magic string definition.
3. Open itself in read-only, binary mode.
4. Find the checksum offset.
5. Compute the checksum.
6. Compare the embedded checksum with the computed checksum.
7. That's it!

For Solaris, high-resolution timer checks were added to see how long the process of self-validation takes. For this small test application the time is negligible (see Section 2.6 for more on this topic). Please see Section 4.4 for a full listing of source code.

2.5 Putting It All Together

To give this technique a try, extract the archive to a working directory and simply type **`make`** (Linux users type **`make –f Makefile.linux`**) at the command prompt. The Makefile will build the embedding application (**`embed`**), test application (**`test`**), one-shot threaded test application (**`test1`**), and a generic MD5 computation utility (**`tmd5`**). Once everything is built, run the test application; it should fail:

```
% ./test
./test: calculation time: 1984574 nsec
43304445424545464241424546414345
./test: checksum verify failed
```

Using **`od`** (`c` and `x` options) to dump the file it's easy to find the embedded magic string (see Section 4.2):

```
0022020    4330    4445    4245    4546    4241    4245    4641    4345
           C   0   D   E   B   E   E   F   B   A   B   E   F   A   C   E
```

Now, embed the checksum by running the embedding application:

```
% ./embed test
./embed: checksum_offset = 9232
./embed: checksum: 6156a6e3d6b1ef27194aac7a357fedee
./embed: verify: 6156a6e3d6b1ef27194aac7a357fedee
./embed: checksum verified
./embed: MD5 (from within file): 6156a6e3d6b1ef27194aac7a357fedee
./embed: true md5: 398a4c1ed5a008d6ec464cf7749f43db
```

And repeat running the test application:

```
% ./test
./test: calculation time: 1841487 nsec
6156a6e3d6b1ef27194aac7a357fedee
./test: checksum verified
```

Note that the checksums from steps 2 and 3 match as they should. Now use **`od`** again (`-x` option only) to verify by hand:

```
0022020 6156 a6e3 d6b1 ef27 194a ac7a 357f edee
```

Repeat the process for the one-shot threaded version (described in Section 2.6.2):

```
% ./test1
main loop running...
validate_thread_: calculation time: 2253491 nsec
43304445424545464241424546414345
validate_thread_: checksum verify failed
```

Embed the checksum:

```
% ./embed test1
./embed: checksum_offset = 9705
./embed: checksum: aa7d11f75d83654efb626960f097e52f
./embed: verify: aa7d11f75d83654efb626960f097e52f
./embed: checksum verified
./embed: MD5 (from within file): aa7d11f75d83654efb626960f097e52f
./embed: true md5: 0e5ba974a59f450d8c732f22c75eb01d
```

And run the test application:

```
% ./test1
main loop running...
validate_thread_: calculation time: 2139716 nsec
aa7d11f75d83654efb626960f097e52f
validate_thread_: checksum verified
main loop running...
main loop running...
main loop running...
main loop running...
^C
```

Now, for extra validation of the concept, try stripping **`test`** and re-running it to see what happens:

```
% strip test
% ./test
./test: calculation time: 1795441 nsec
5d1aab77eef1b488a88a91c7e6f931e9
./test: checksum verify failed
```

In this instance `test` fails as expected because stripping the executable changed its contents, thereby altering the MD5 sum. Similar results may be expected by repeating the process for **`test1`**:

```
% strip test1
./test1
main loop running...
validate_thread_: calculation time: 2083831 nsec
105ad3519e8db11ddae3ddce6b6ed8b2
validate_thread_: checksum verify failed
```

The Makefile provides additional targets of interest: clean and release. As expected, clean removes all objects and executables. Release simulates a release environment by first performing the strip enhancement and then by embedding the checksum so that the test executables are ready to go directly after building.

©2005 Dot4, Inc.

2.6 Variations

The above example is a fairly rudimentary implementation of this concept: the application validates itself by running the check once at startup. While this may be practical for smaller applications (where perhaps this technique is generally needed less), it is probably not practical for larger applications (>25MB) for a few related reasons.

First, the sheer size of a larger application directly impacts the length of time required to compute the requisite MD5 sums. For application sizes in the range of tens-of-megabytes and greater, using this method could result in the application taking several seconds to crunch the numbers to compute the MD5 sum.

Second, during this time the application is doing nothing else, which makes it much easier to determine what the application is doing and therefore hack it or otherwise defeat the validation mechanism.

Third, users will have to wait longer to begin using the application once it is started, which can be frustrating in the "gotta have it now" day and age.

It therefore makes sense to tune the MD5 engine's computation buffer size to the application size. A series of quick experiments yielded useful data for tuning the MD5 buffer size. The table below shows the MD5 computation time for four file sizes and eight buffer sizes. The files sizes were (approximately): 512kB, 1MB, 15MB, and 50MB. The buffer sizes were 1kB, 2kB, 4kB, 8kB, 16kB, 32kB, 64kB, and 128kB. The test was conducted on a Sun Ultra5 Workstation with a 333MHz Sparc processor and 512MB of RAM with minimal loading. Graphical representations follow the table. Note that for the larger files, the sweet spot lies around a 16kB to 32kB buffer size, while for the smaller files it's around 8kB to 16kB. For the purposes of using this technique on a larger file a 16kB or 32kB buffer would be used for two reasons: first, computation times start increasing, albeit slowly, for larger files above a 32kB buffer size; second, once the buffer size is greater than 8kB the highest percentage of time gain versus buffer size is already achieved.

With this in mind, the following variations on this technique are proffered.

File Size	Buffer Size (bytes)	Time (nanoseconds)
512kB	1024	64771117
	2048	25956317
	4096	34336114
	8192	30976730
	16384	22703027
	32768	33424836
	65536	43338807
	131072	40483639
1MB	1024	147553676
	2048	76958702
	4096	102762970
	8192	80092463
	16384	90150887
	32768	90386085
	65536	104375875
	131072	93108133
15MB	1024	1746431038
	2048	1378312690
	4096	1327530608
	8192	1232254652
	16384	1219765048
	32768	1192526891
	65536	1196725806
	131072	1206872334
50MB	1024	5737609641
	2048	4595211009
	4096	4261892347
	8192	4167450515
	16384	4021871353
	32768	4073216417
	65536	4068552781
	131072	4083544255

Table 1 - Buffer Size vs. Time

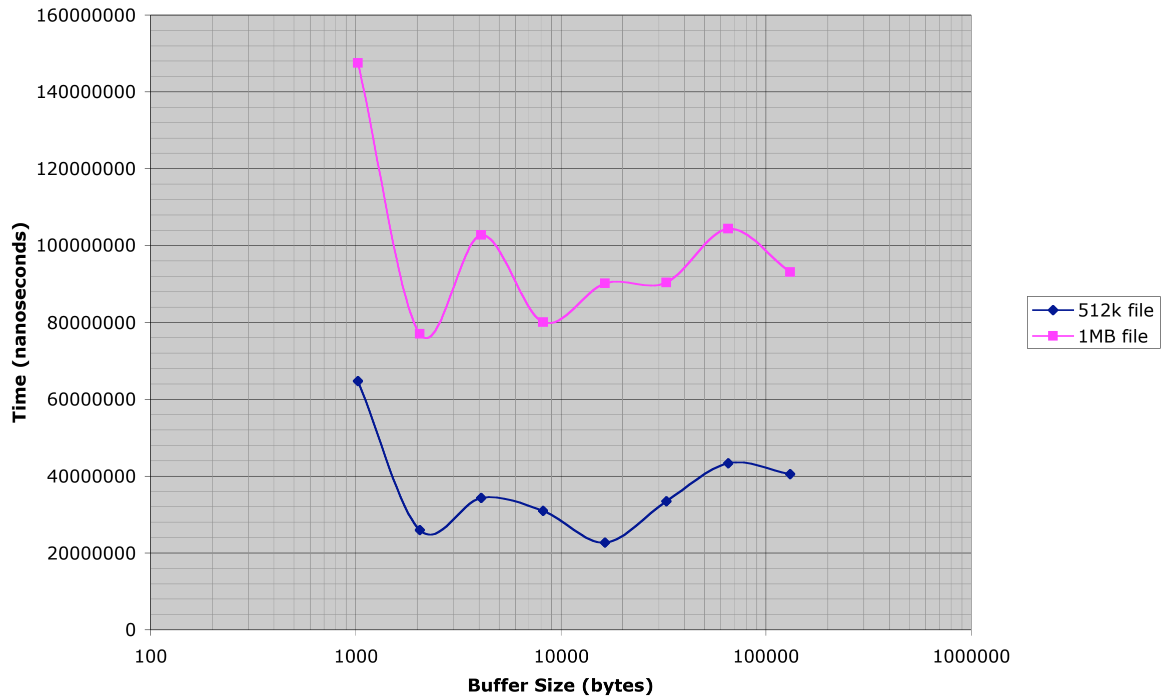

Figure 1 – Small File MD5 Computation Time vs. Buffer Size

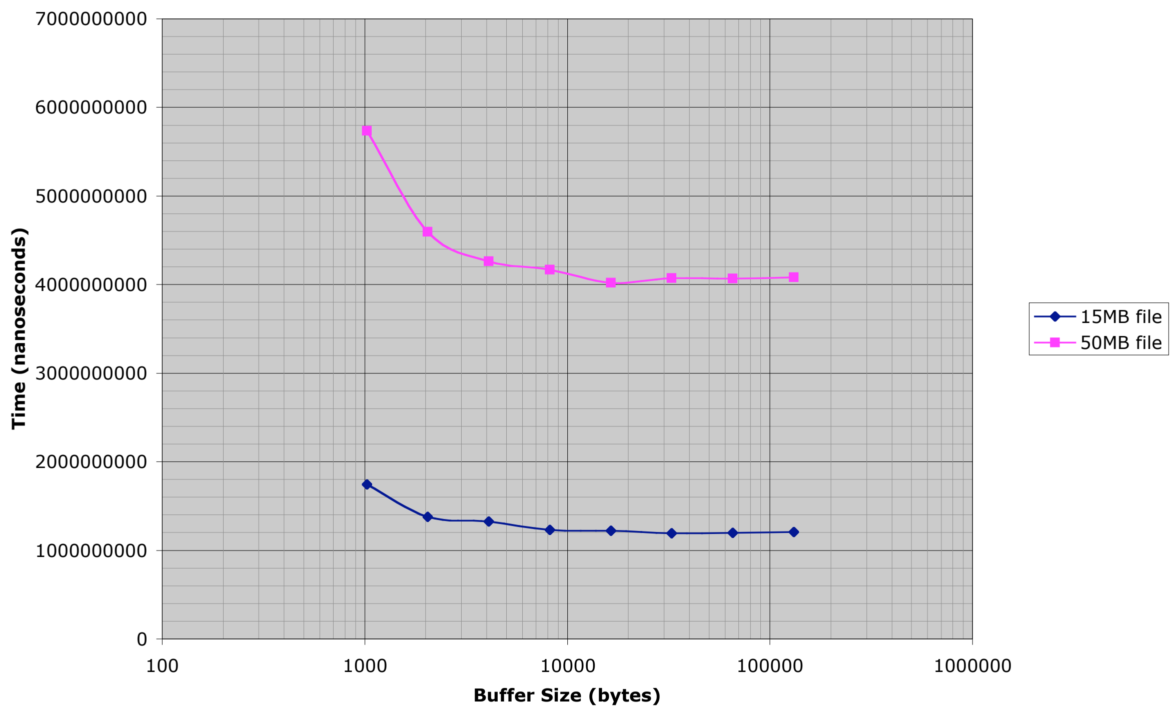

Figure 2 – Large File MD5 Computation Time vs. Buffer Size

2.6.1 Split Checksums

The Split Checksums method is a slight twist on the original theme. Instead of using a single MD5 sum, use two or more with checks in different portions of executable code. That way the validation is performed at different times and in different places thereby making it more difficult to defeat. This method requires modifying `MD5_csum_offset_get()` and `MD5_embedded_csum_calc()` or splitting them into two or more routines based on how many checksums are embedded. No source code is given for this variation.

2.6.2 One-shot Threaded

The One-shot Threaded method is more appropriate to larger applications where startup blocking is not desirable but only an initial validation is required. In this case, create a thread at startup and have it perform the necessary checks. If the checks fail, the thread can handle cleanup and perform a graceful exit. While not required, it is good practice to set the thread priority appropriately even though it is short-lived. This method requires no changes to the MD5 engine. Source code for this version of the test application is shown in Section 4.4.1.

2.6.3 Continuous Threaded

The Continuous Threaded method is similar to the One-Shot method except that the checks are performed over and over during program execution. Periodic checks can be performed without interruption or even randomly based on a timer interval set in either `select()` or `poll()`. Again, if at any time a check fails, the thread can perform the appropriate cleanup and then gracefully exit. Note that using this method carries two caveats. First, care must be taken to avoid virtual memory thrashing due to constant loading of new pages for checksumming. Second, be sure to set thread priorities appropriately. The validation thread must be scheduled low enough so that it does not affect the main application processing in a negative manner (or cause thrashing as mentioned above), especially if it is real-time, but also high enough so that is not processor starved, thereby defeating its purpose. Experimentation on your specific application and system will yield the best balance. This method also requires no changes to the MD5 engine. No source code is given for this variation.

2.7 Enhancements

This technique, like all other software security techniques, is one technique of many that can be employed to help prevent unauthorized access to or modification of an application. No single technique can provide absolute security and the selection of a technique must be weighed on its strengths and weaknesses against the potential intrusions that are known and those that are thought to exist. To be clear, the author is not an encryption or security expert, nor claims to be. That being said, there are some things an application designer can do to enhance the effectiveness of using embedded checksums for application self-validation. Since no custom algorithms are presented here, all of these enhancements basically boil down to removing information whose unintended purpose would be to assist hackers in defeating the validation mechanism. No source code examples are given for these enhancements.

First, use only release executables that have been stripped and contain no debugging information whatsoever. Several Unix utilities exist for easily prying into ELF files and the more limited information these display about your application, the better.

Second, obfuscate the MD5 engine function names. This will prevent prying eyes from knowing the names of library functions when dumping the raw contents of the application. A simple technique for this is to use a macro for the function name and let the preprocessor take care of the rest. For example, assume you want to obfuscate the function `MD5_csum_offset_get()`. The obfuscated name doesn't all that much matter and could simply be generated randomly by your brain. Alternately, you could create a small application to create a hashed name and then use that. The only requirement is that all obfuscated names be unique. Now simply create the following example macro before the original function prototype in `md_5.h` and then rebuild:

```
#define MD5_csum_offset_get    6a44e5f02eec
```

Nothing else needs to change. Once compiled, doing a hexdump on the application and searching for strings will yield nothing indicative about the purpose of the functions that are compiled in. As part of this technique also consider obfuscating any validation thread names as well as the name of the magic string global variable(s).

Third, enhance function obfuscation as described above by creating an intermediate function table. Using a function table changes the assembly listing (as possibly generated by a disassembler) by not referencing the desired function directly by name at the calling point. It instead makes that connection when the table is assigned, which can be in a very different location of the program. In a similar manner, consider making access to the embedded MD5 sum only available through an accessor function.

The important thing to remember when using these techniques is that ultimately whether or not a modified application runs boils down to one or more conditional statements. It is goal of the designer, and beyond the scope of this paper, to prevent would-be intruders from finding and overcoming them all and limiting the damage that they can cause.

©2005 Dot4, Inc.

3 Conclusion

A novel way to perform ELF application self-validation has been described along with several variations and enhancements. These variations and enhancements can increase the flexibility of this technique when applying it against other unique applications. Additionally, design optimizations were presented to maximize benefit and reduce computational load. Last, source code and references are presented following this section.

4 Appendix – Source Code

The source code for this white paper is published under the GNU General Public License (GPL) and is freely downloadable from the Internet[7]. Only the source modules relevant to this white paper have been reproduced here.

4.1 MD5 Engine

4.1.1 md5_.h

```
/*
** Copyright (c) 2005
**
** Dot4 Corporation/Todd M. Mizenko
** 6080 Jericho Turnpike, Suite 211
** Commack, NY 11725
** 800/834-1951
** http://www.dot4.com
**
** This software is free software; you can redistribute it and/or
** modify it under the terms of the GNU General Public License
** as published by the Free Software Foundation; either version 2 of
** the License, or (at your option) any later version.
**
** This program is distributed in the hope that it will be useful, but
** WITHOUT ANY WARRANTY; without even the implied warranty of
** MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
** General Public License for more details.
**
** You should have received a copy of the GNU General Public License
** along with this program; if not, write to the Free Software
** Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
*/


#ifndef MD5_H_INCLUDED
#define MD5_H_INCLUDED


#define D_MD5_SIZE 16


int MD5_csum_offset_get(int fd, const char *magic_str, int magic_size);
int MD5_embedded_csum_calc(int fd, unsigned int checksum_offset,
                           unsigned char *checksum);
int MD5_checksum_calc(int fd, unsigned char *checksum);
int MD5_checksum_embed(int fd, int offset, const unsigned char *checksum);
int MD5_checksum_print(const unsigned char *checksum);
int MD5_buffer_size_set(int buf_size);
int MD5_buffer_size_get(void);


#endif /* MD5_H_INCLUDED */
```

4.1.2 md5.c

```
/*
** Copyright (c) 2005
**
** Dot4 Corporation/Todd M. Mizenko
** 6080 Jericho Turnpike, Suite 211
** Commack, NY 11725
** 800/834-1951
** http://www.dot4.com
**
** This software is free software; you can redistribute it and/or
** modify it under the terms of the GNU General Public License
** as published by the Free Software Foundation; either version 2 of
** the License, or (at your option) any later version.
**
** This program is distributed in the hope that it will be useful, but
** WITHOUT ANY WARRANTY; without even the implied warranty of
** MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
** General Public License for more details.
**
** You should have received a copy of the GNU General Public License
** along with this program; if not, write to the Free Software
** Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
*/


#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <unistd.h>
#include <errno.h>
#ifdef SOLARIS
#include <libelf.h>
#endif
#ifdef LINUX
#include <elf.h>
#include <md5global.h>
#endif
#include <md5.h>

#include "md5_.h"


#define D_BUF_SIZE 16384


/* ---------------------------------------------------------------------------
** MD5_csum_offset_get()
**
** This function calculates the offset into the given file (specified by fd)
** of the magic checksum string.
** ---------------------------------------------------------------------------
*/
int
MD5_csum_offset_get(int fd, const char *magic_str, int magic_size)
{
```

```
const char func[] = "MD5_csum_offset_get";
char *string_table;
char *rodata;
char *magic_ptr;
int str_tab_size;
int idx;
int size;
int offset;
int checksum_offset;
Elf32_Ehdr elf_hdr;
Elf32_Shdr *sect_hdr;

        if(fd < 0) {
                printf("%s: negative fd (%d)\n", func, fd);
                return(-1);
        }

        if((!magic_str) || (!magic_size)) {
                printf("%s: invalid magic\n", func);
                return(-1);
        }

/*
** Read in the main ELF header and check to see that it is valid
** and contains the target type of ELF file.
*/
        if(lseek(fd, 0, SEEK_SET) != 0) {
                printf("%s: lseek() to 0 error\n", func);
                return(-1);
        }

        if(read(fd, &elf_hdr, sizeof(Elf32_Ehdr)) != sizeof(Elf32_Ehdr)) {
                printf("%s: read() elf_hdr error\n", func);
                return(-1);
        }

        if(strncmp((const char *)elf_hdr.e_ident, ELFMAG, SELFMAG) != 0) {
                printf("%s: strncmp() elf magic error\n", func);
                return(-1);
        }

        if((elf_hdr.e_type != ET_EXEC) || (!(elf_hdr.e_version)) ||
           (elf_hdr.e_ident[EI_CLASS] != ELFCLASS32)) {
                printf("%s: invalid elf file\n", func);
                return(-1);
        }

/*
** Compute the size of the section header table, malloc a buffer for it,
** seek to its location, and read in the section header table.
*/
        size = elf_hdr.e_shnum * sizeof(Elf32_Shdr);
        if((sect_hdr = (Elf32_Shdr *)malloc(size)) == NULL) {
                printf("%s: malloc() sect_hdr error\n", func);
                return(-1);
        }

        if(lseek(fd, elf_hdr.e_shoff, SEEK_SET) != elf_hdr.e_shoff) {
                printf("%s: lseek() sect_hdr error\n", func);
```

```
                return(-1);
        }

        if(read(fd, sect_hdr, size) != size) {
                printf("%s: read() sect_hdr error\n", func);
                return(-1);
        }

/*
** Next, load in the string table to help us locate which section .rodata is in.
** The string table is indexed by the e_shstrndx member of the ELF header.
** Similar routine: get the string table size, malloc a buffer for it,
** seek to its location, and read it in.
*/
        str_tab_size = sect_hdr[elf_hdr.e_shstrndx].sh_size;
        if((string_table = calloc(str_tab_size, 1)) == NULL) {
                printf("%s: calloc() string_table error\n", func);
                return(-1);
        }

        if(lseek(fd, sect_hdr[elf_hdr.e_shstrndx].sh_offset, SEEK_SET) !=
           sect_hdr[elf_hdr.e_shstrndx].sh_offset) {
                printf("%s: lseek() string_table error\n",
                       func);
                return(-1);
        }

        if(read(fd, string_table, str_tab_size) != str_tab_size) {
                printf("%s: read() string_table error\n", func);
                return(-1);
        }

/*
** Now we search the string table for ".rodata". Once we find it, we then
** what section to index into.
*/
        for(idx=0; idx < elf_hdr.e_shnum; idx++) {
                if(strncmp((string_table + sect_hdr[idx].sh_name),
                           ".rodata", 7)==0) {
/*
** Seek to the .rodata section, malloc enough space for it, then read it in.
*/
                        if(lseek(fd, sect_hdr[idx].sh_offset, SEEK_SET) !=
                           sect_hdr[idx].sh_offset) {
                                printf("%s: lseek() rodata error\n", func);
                                return(-1);
                        }

                        if((rodata = malloc(sect_hdr[idx].sh_size)) == NULL) {
                                printf("%s: malloc() rodata error\n", func);
                                return(-1);
                        }

                        if(read(fd, rodata, sect_hdr[idx].sh_size) !=
                           sect_hdr[idx].sh_size) {
                                printf("%s: read() rodata error\n", func);
                                return(-1);
                        }
```

```
/*
** Start at the beginning of the .rodata section and increment character by
** character until we find the magic string we are looking for. Once we have
** found it, compute the absolute offset into the file, where the checksum
** will be embedded.
*/
                        magic_ptr = rodata;

                        while(magic_ptr < (rodata + sect_hdr[idx].sh_size)) {
                                if(strncmp(magic_ptr, magic_str, magic_size) ==
                                    0) {
                                offset = magic_ptr - rodata;
                                checksum_offset=sect_hdr[idx].sh_offset+offset;
                                break;
                                }
                                magic_ptr++;
                        }

                        free(rodata);
                        free(string_table);
                        free(sect_hdr);

/*
** if we scanned the whole .rodata section and didn't find our magic string
** then something went horribly awry... otherwise, return the checksum offset.
*/
                        if(magic_ptr >= (string_table+sect_hdr[idx].sh_size)) {
                                return(-1);
                        }
                        else {
                                return checksum_offset;
                        }
                }
        }

        free(rodata);
        free(string_table);
        free(sect_hdr);

        return (-1);
}


/* ---------------------------------------------------------------------------
** MD5_embedded_csum_calc()
**
** This function computes the checksum for a file, skipping the area
** specified by checksum offset. This area is typically used for embedding
** a checksum into an executable after compilation.
** ---------------------------------------------------------------------------
*/
int
MD5_embedded_csum_calc(int fd, unsigned int checksum_offset,
                       unsigned char *checksum)
{
const char func[] = "MD5_embedded_csum_calc";
unsigned char *md5in;
int buf_size;
int rc;
```

```
int read_size;
int total_bytes;
MD5_CTX md5_context;

/*
** validate the file descriptor, rewind the file to the beginning, and
** malloc our read buffer.
*/
        if(fd < 0) {
                printf("%s: fd (%d) < 0\n", func, fd);
                return(-1);
        }

        if((checksum_offset == 0) || (!checksum)) {
                printf("%s: invalid checksum offset/checksum buffer\n", func);
                return(-1);
        }

        if(lseek(fd, 0, SEEK_SET) != 0) {
                printf("%s: lseek() 1 failed rc=%d, errno=%d (%s)\n",
                        func, rc, errno, strerror(errno));
                return(-1);
        }

        buf_size = MD5_buffer_size_get();
        if((md5in = (unsigned char *)malloc(buf_size)) == NULL) {
                printf("%s: malloc() failed\n", func);
                return(-1);
        }

/*
** initialize our md5 context and check for a checksum offset that's less
** than our read buffer size. if the offset is smaller, then adjust the
** initial read size accordingly.
*/
        MD5Init(&md5_context);

        if(checksum_offset < buf_size) {
                read_size = checksum_offset;
        }
        else {
                read_size = buf_size;
        }

/*
** this first read/md5 update chunk loops from the start
** of the file to the location of the embedded checksum.
*/
        total_bytes = 0;
        while(total_bytes <= checksum_offset) {
                if((rc = read(fd, md5in, read_size)) < 0) {
                        if((errno == EINTR) || (errno == EAGAIN)) {
                                continue;
                        }
                        else {
                                printf("%s: read failed (%d): errno=%d\n",
                                        func, rc, errno);
                                return(-1);
                        }
```

```
		}
		else if(rc == 0) {
			/* nothing left to read... */
			break;
		}

		MD5Update(&md5_context, md5in, rc);

		total_bytes += rc;
		if((checksum_offset - total_bytes) < buf_size) {
			read_size = checksum_offset - total_bytes;
		}
	}

/*
** next, seek to the first location past the embedded checksum
*/
	if(lseek(fd, (checksum_offset + D_MD5_SIZE), SEEK_SET) !=
	   (checksum_offset + D_MD5_SIZE)) {
		printf("%s: lseek() 2 failed rc=%d, errno=%d (%s)\n",
		       func, rc, errno, strerror(errno));
		return(-1);
	}

/*
** now, loop from this point to the end of the file, updating
** the md5 along the way.
*/
	while(1) {
		if((rc = read(fd, md5in, read_size)) < 0) {
			if((errno == EINTR) || (errno == EAGAIN)) {
				continue;
			}
			else {
				printf("%s: read failed (%d): errno=%d\n",
				       func, rc, errno);
				return(-1);
			}
		}
		else if(rc == 0) {
			/* nothing left to read... */
			break;
		}

		MD5Update(&md5_context, md5in, rc);
	}

/*
** finally, finsh the md5 and get out.
*/
	MD5Final(checksum, &md5_context);

	return 0;
}


/* ----------------------------------------------------------------------------
** MD5_checksum_calc()
**
```

©2005 Dot4, Inc.

```
** This function computes a traditional MD5 sum over the file specified by
** the given file descriptor.
** ----------------------------------------------------------------------------
*/
int
MD5_checksum_calc(int fd, unsigned char *checksum)
{
const char func[] = "MD5_checksum_calc";
unsigned char *md5in;
int buf_size;
int rc;
int read_size;
MD5_CTX md5_context;

/*
** validate the file descriptor, rewind the file to the beginning, and
** malloc our read buffer.
*/
	if(fd < 0) {
		printf("%s: fd (%d) < 0\n", func, fd);
		return(-1);
	}

	if(lseek(fd, 0, SEEK_SET) != 0) {
		printf("%s: lseek() failed rc=%d, errno=%d (%s)\n",
			func, rc, errno, strerror(errno));
		return(-1);
	}

	buf_size = MD5_buffer_size_get();
	if((md5in = (unsigned char *)malloc(buf_size)) == NULL) {
		printf("%s: malloc() failed\n", func);
		return(-1);
	}

/*
** initialize the context and read/update until the end of the file.
** finalize the checksum and get out.
*/
	MD5Init(&md5_context);
	read_size = buf_size;

	while(1) {
		if((rc = read(fd, md5in, read_size)) < 0) {
			if((errno == EINTR) || (errno == EAGAIN)) {
				continue;
			}
			else {
				printf("%s: read failed (%d): errno=%d\n",
					func, rc, errno);
				return(-1);
			}
		}
		else if(rc == 0) {
			/* nothing left to read... */
			break;
		}

		MD5Update(&md5_context, md5in, rc);
```

```
        }

        MD5Final(checksum, &md5_context);

        return 0;
}


/* ---------------------------------------------------------------------------
** MD5_checksum_embed()
**
** This function embeds an MD5 checksum at the provided offset.
** ---------------------------------------------------------------------------
*/
int
MD5_checksum_embed(int fd, int offset, const unsigned char *checksum)
{
const char func[] = "MD5_checksum_embed";
int rc;

        if(fd < 0) {
                printf("%s: fd (%d) < 0\n", func, fd);
                return(-1);
        }

        if((!offset) || (!checksum)) {
                printf("%s: invalid offset/checksum buffer\n", func);
                return(-1);
        }

        if((rc = lseek(fd, offset, SEEK_SET)) != offset) {
                printf("%s: lseek() failed, rc=%d, errno=%d (%s)\n",
                       func, rc, errno, strerror(errno));
                return(-1);
        }

        if(write(fd, checksum, D_MD5_SIZE) < 0) {
                printf("%s: write() failed\n", func);
                return(-1);
        }

        return(0);
}


/* ---------------------------------------------------------------------------
** MD5_checksum_extract()
**
** This function extracts a previously embedded MD5 sum.
** ---------------------------------------------------------------------------
*/
int
MD5_checksum_extract(int fd, int offset, unsigned char *checksum)
{
const char func[] = "MD5_checksum_extract";
int rc;

        if(fd < 0) {
                printf("%s: fd (%d) < 0\n", func, fd);
```

```
                return(-1);
        }

        if((!offset) || (!checksum)) {
                printf("%s: invalid offset/checksum buffer\n", func);
                return(-1);
        }

        if((rc = lseek(fd, offset, SEEK_SET)) != offset) {
                printf("%s: lseek() failed, rc=%d, errno=%d (%s)\n",
                       func, rc, errno, strerror(errno));
                return(-1);
        }

        if(read(fd, checksum, D_MD5_SIZE) < 0) {
                printf("%s: read() failed\n", func);
                return(-1);
        }

        return(0);
}


/* ----------------------------------------------------------------------------
** MD5_checksum_print()
**
** This function simply prints a previously calculated MD5 sum.
** ----------------------------------------------------------------------------
*/
int
MD5_checksum_print(const unsigned char *checksum)
{
const char func[] = "MD5_checksum_print";
int idx;

        if(!checksum) {
                printf("%s: NULL checksum\n", func);
                return(-1);
        }

        for(idx=0; idx < D_MD5_SIZE; idx++) {
                printf("%02x", checksum[idx]);
        }
        printf("\n");
}


static int G_buffer_size_ = D_BUF_SIZE;


/* ----------------------------------------------------------------------------
** MD5_buffer_size_set()
**
** This function sets the MD5 engine calculation buffer size. It is used to
** fine tune the MD5 processing for specific file sizes. It returns the old
** buffer size or -1 if buf_size is less than 0.
**
** ----------------------------------------------------------------------------
*/
```

```
int
MD5_buffer_size_set(int buf_size)
{
int old_buf_size = G_buffer_size_;

        if(buf_size < 0) {
                return -1;
        }

        G_buffer_size_ = buf_size;
        return old_buf_size;
}


/* ----------------------------------------------------------------------------
** MD5_buffer_size_get()
**
** This function gets the MD5 engine calculation buffer size.
**
** ----------------------------------------------------------------------------
*/
int
MD5_buffer_size_get(void)
{
        return G_buffer_size_;
}
```

4.2 Common Header – magic.h

```
/*
** Copyright (c) 2005
**
** Dot4 Corporation/Todd M. Mizenko
** 6080 Jericho Turnpike, Suite 211
** Commack, NY 11725
** 800/834-1951
** http://www.dot4.com
**
** This software is free software; you can redistribute it and/or
** modify it under the terms of the GNU General Public License
** as published by the Free Software Foundation; either version 2 of
** the License, or (at your option) any later version.
**
** This program is distributed in the hope that it will be useful, but
** WITHOUT ANY WARRANTY; without even the implied warranty of
** MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
** General Public License for more details.
**
** You should have received a copy of the GNU General Public License
** along with this program; if not, write to the Free Software
** Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
*/


#ifndef MAGIC_H_INCLUDED
#define MAGIC_H_INCLUDED


#define D_MAGIC_SIZE        17
#define D_MAGIC_STRING      "C0DEBEEFBABEFACE"


#endif /* MAGIC_H_INCLUDED */
```

4.3 Embedding Application – embed.c

```
/*
** Copyright (c) 2005
**
** Dot4 Corporation/Todd M. Mizenko
** 6080 Jericho Turnpike, Suite 211
** Commack, NY 11725
** 800/834-1951
** http://www.dot4.com
**
** This software is free software; you can redistribute it and/or
** modify it under the terms of the GNU General Public License
** as published by the Free Software Foundation; either version 2 of
** the License, or (at your option) any later version.
**
** This program is distributed in the hope that it will be useful, but
** WITHOUT ANY WARRANTY; without even the implied warranty of
** MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
** General Public License for more details.
**
** You should have received a copy of the GNU General Public License
** along with this program; if not, write to the Free Software
** Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
*/


#include <unistd.h>
#ifdef SOLARIS
#include <libelf.h>
#endif
#ifdef LINUX
#include <elf.h>
#endif
#include <stdlib.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>


#include "md5_.h"
#include "magic.h"


int
main(int argc, char *argv[])
{
const char *func = argv[0];
unsigned char checksum[D_MD5_SIZE] = "";
unsigned char verify[D_MD5_SIZE] = "";
int fd;
int rc;
int checksum_offset;

/*
** basic validation: we need an open file before we can do anything.
*/
```

```
        if(argv[1] == NULL) {
                printf("%s: no input file specified\n", func);
                return(-1);
        }

        if((fd = open(argv[1], O_RDWR)) < 0) {
                printf("%s: could not open file %s\n", func, argv[1]);
                return(-1);
        }

/*
** go get the checksum offset buried in the .rodata section.
*/
        if((checksum_offset = MD5_csum_offset_get(fd, D_MAGIC_STRING,
                                            (D_MAGIC_SIZE-1))) < 0) {
                printf("%s: MD5_csum_offset_get() failed\n", func);
                return(-1);
        }

        printf("%s: checksum_offset = %d\n", func, checksum_offset);

/*
** calculate the MD5 checksum around the space embedded in the file,
** then embed the checksum into the file.
*/
        if(MD5_embedded_csum_calc(fd, checksum_offset, checksum) < 0) {
                printf("%s: MD5_embedded_csum_calc() 1 failed\n", func);
                return(-1);
        }

        printf("%s: checksum: ", func);
        MD5_checksum_print(checksum);

        if(MD5_checksum_embed(fd, checksum_offset, checksum) < 0) {
                printf("%s: MD5_checksum_embed() failed\n", func);
                return(-1);
        }

/*
** verify that we didn't mess anything up by re-calculating the MD5 and
** comparing to the first pass. in theory, if the checksum isn't in the
** correct place we'll get back a different value from the first pass.
*/
        if(MD5_embedded_csum_calc(fd, checksum_offset, verify) < 0) {
                printf("%s: MD5_embedded_csum_calc() 2 failed\n", func);
                return(-1);
        }

        printf("%s: verify: ", func);
        MD5_checksum_print(verify);

        if(memcmp(checksum, verify, D_MD5_SIZE) != 0) {
                printf("%s: memcmp failed\n", func);
        }
        else {
                printf("%s: checksum verified\n", func);
        }

/*
```

```
** see what we actually embedded into the file.
*/
        if(MD5_checksum_extract(fd, checksum_offset, checksum) < 0) {
                printf("%s: MD5_checksum_extract() failed\n", func);
                return(-1);
        }

        printf("%s: MD5 (from within file): ", func);
        MD5_checksum_print(checksum);

/*
** biff out the normal checksum for kicks.
*/
        if(MD5_checksum_calc(fd, checksum) < 0) {
                printf("%s: MD5_checksum_calc() failed\n", func);
                return(-1);
        }

        printf("%s: true md5: ", func);
        MD5_checksum_print(checksum);

        close(fd);
}
```

4.4 Test Application – test.c

```
/*
** Copyright (c) 2005
**
** Dot4 Corporation/Todd M. Mizenko
** 6080 Jericho Turnpike, Suite 211
** Commack, NY 11725
** 800/834-1951
** http://www.dot4.com
**
** This software is free software; you can redistribute it and/or
** modify it under the terms of the GNU General Public License
** as published by the Free Software Foundation; either version 2 of
** the License, or (at your option) any later version.
**
** This program is distributed in the hope that it will be useful, but
** WITHOUT ANY WARRANTY; without even the implied warranty of
** MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
** General Public License for more details.
**
** You should have received a copy of the GNU General Public License
** along with this program; if not, write to the Free Software
** Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
*/


#include <fcntl.h>
#include <stdio.h>
#ifdef SOLARIS
#include <sys/time.h>
#endif


#include "md5_.h"
#include "magic.h"


static const char magic[D_MAGIC_SIZE] = D_MAGIC_STRING;


int
main(int argc, char *argv[])
{
const char *func = argv[0];
unsigned char checksum[D_MD5_SIZE];
unsigned char verify[D_MD5_SIZE];
int fd;
int offset;
#ifdef SOLARIS
hrtime_t end;
hrtime_t diff;
hrtime_t start;
#endif

        if((fd = open(argv[0], O_RDONLY)) < 0) {
                printf("%s: could not open file %s\n", func, argv[0]);
                return(-1);
        }
```

```
#ifdef SOLARIS
        start = gethrtime();
#endif
        offset = MD5_csum_offset_get(fd, magic, (D_MAGIC_SIZE-1));
        MD5_embedded_csum_calc(fd, offset, checksum);
#ifdef SOLARIS
        end = gethrtime();
        diff = end - start;
        printf("%s: calculation time: %lld nsec\n", func, diff);
#endif

        MD5_checksum_print((const unsigned char *)magic);

        if(memcmp(checksum, magic, D_MD5_SIZE) != 0) {
                printf("%s: checksum verify failed\n", func);
        }
        else {
                printf("%s: checksum verified\n", func);
        }

        close(fd);
        return(0);
}
```

4.4.1 One-shot Threaded – test1.c

```
/*
** Copyright (c) 2005
**
** Dot4 Corporation/Todd M. Mizenko
** 6080 Jericho Turnpike, Suite 211
** Commack, NY 11725
** 800/834-1951
** http://www.dot4.com
**
** This software is free software; you can redistribute it and/or
** modify it under the terms of the GNU General Public License
** as published by the Free Software Foundation; either version 2 of
** the License, or (at your option) any later version.
**
** This program is distributed in the hope that it will be useful, but
** WITHOUT ANY WARRANTY; without even the implied warranty of
** MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
** General Public License for more details.
**
** You should have received a copy of the GNU General Public License
** along with this program; if not, write to the Free Software
** Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
*/


#include <fcntl.h>
#include <stdio.h>
#include <pthread.h>
#include <poll.h>
#ifdef SOLARIS
#include <sys/time.h>
#endif


#include "md5_.h"
#include "magic.h"


static const char magic[D_MAGIC_SIZE] = D_MAGIC_STRING;

static void *validate_thread_(void *arg);


int
main(int argc, char *argv[])
{
pthread_t tid;

        pthread_create(&tid, NULL, validate_thread_, (void *)argv[0]);

        while(1) {
                printf("main loop running...\n");
                poll(0, 0, 1000); /* 1 second */
        }
}
```

```
static void *
validate_thread_(void *arg)
{
const char func[] = "validate_thread_";
char *fname = (char *)arg;
unsigned char checksum[D_MD5_SIZE];
unsigned char verify[D_MD5_SIZE];
int fd;
int offset;
#ifdef SOLARIS
hrtime_t end;
hrtime_t diff;
hrtime_t start;
#endif

        if((fd = open(fname, O_RDONLY)) < 0) {
                printf("%s: could not open file %s\n", func, fname);
                exit(-1);
        }

#ifdef SOLARIS
        start = gethrtime();
#endif
        offset = MD5_csum_offset_get(fd, magic, (D_MAGIC_SIZE-1));
        MD5_embedded_csum_calc(fd, offset, checksum);
#ifdef SOLARIS
        end = gethrtime();
        diff = end - start;
        printf("%s: calculation time: %lld nsec\n", func, diff);
#endif

        MD5_checksum_print((const unsigned char *)magic);

        if(memcmp(checksum, magic, D_MD5_SIZE) != 0) {
                printf("%s: checksum verify failed\n", func);
                /* perform any necessary cleanup here... */
                exit(-1);
        }
        else {
                printf("%s: checksum verified\n", func);
        }

        close(fd);
        return;
}
```

5 References

One other inspiration for this idea was Matt Fisher, “Protecting Binary Executables”, Embedded Systems Programming, February 2000.

---

[1] http://en.wikipedia.org/wiki/Main_Page
[2] RFC describing the MD5 algorithm: http://www.ietf.org/rfc/rfc1321.txt
[3] Wikipedia discussion of MD5: http://en.wikipedia.org/wiki/MD5
[4] The Intel ELF specification is available online from http://www.wotsit.org/download.asp?f=elf11g
[5] Eric Youngdale, “The ELF Object File Format: Introduction”, Linux Journal, April 1995, http://www.linuxjournal.com/article/1059; “The ELF Object File Format”, Linux Journal, May 1995, http://www.linuxjournal.com/article/1060
[6] When researching ELF parsing the following link came up: http://archives.neohapsis.com/archives/vuln-dev/2002-q1/att-1130/01-fuzztest.c. The parser presented here is based on this work.
[7] http://www.dot4.com/literature.html