Fortify Software

NIST SHA-3 Competition Security Audit Results

Joy Forsythe, Security Researcher
Douglas Held, Software Security Consultant

Abstract
The National Institutes of Standards and Technology ("NIST") is holding a competition to choose a design for the Secure Hash Algorithm version 3 ("SHA-3"). The reference implementations of some of the contestants have bugs in them that could cause crashes, performance problems or security problems if they are used in their current state. Based on our bug reports, some of those bugs have already been fixed.

1 Introduction

The inspiration for the project was the result of testing of a pre-release version of Fortify SCA against the Skein and MD6 reference implementations. This was prompted by an article introducing the NIST round 1 entries on the technology discussion website Slashdot.org[1]. The original idea was to test complex C code that was unlikely to contain defects.

We were surprised to discover buffer overflows in the MD6 implementation. We carefully reviewed the automated results and contacted the author, Professor Ron Rivest. The MD6 team to confirmed the findings and they resubmitted a corrected version of the implementation to NIST. Based on this positive outcome, we decided to do a similar review of the remaining SHA-3 submittals.

Ultimately two projects, MD6 and Blender, contained buffer overflows. The remainder of the issues included out-of-bounds reads, memory leaks and null dereferences, mostly artifacts of incomplete error handling. The null dereferences resulted from failures to check the result of memory allocation; the memory leaks were failures to free resources when handling certain error conditions.

The quality of the code was good overall, but it's important for the reference implementations to be correct. Reference implementations do form the basis for real implementations, and we should reasonably expect them to be used "as is"[3].

Bugs such as these have the potential to affect performance, a key determinant of the NIST competition. For example, under-allocating a buffer can skew the cost estimate for an embedded system. Additional guards against error conditions cost processor cycles. In that sense, bugs do have the potential to affect the outcome of the competition.

Overall, the intent is not to judge any algorithm based on some implementation errors, but for the errors to be corrected so that 1) the best algorithm can be selected and 2) to prevent propagation of mistakes into production code.

2 Methodology

2.1 Analysis

Each entry to the competition posted a submission on the NIST Round 1 webpage[4]. We downloaded the most up-to-date submission packages (as of February 12, 2009) and attempted to build the code found in the "Reference Implementation" directory of each of the 43 projects[5]. Fortify SCA requires "compileable" code in order to invoke the C / C++ preprocessor and translate the code into an intermediate model. In one case, we did need to download and include NIST's genKAT.c file[6] to compile the project.

Whenever possible, the analysis was performed on a Mac OS X 64 bit machine with GCC 4.0.1 (i686-apple-darwin9-gcc-4.0.1). The equivalent build command used was either 'make' in the case of a Makefile, or

```
gcc -c *.c
```

whenever no Makefile or other build instructions were included.

A number of projects required Microsoft Visual Studio and were generally compiled on a Microsoft Windows XP 32 bit machine with Microsoft Visual Studio 2005. In one case, Visual Studio x64 was required and the Microsoft Visual Studio 2008 x64 cross compilation environment was used on the 32 bit operating system.

Upon translation into the Fortify intermediate model, the projects were analyzed against the Fortify 2008-Q4 rules, with a pre-release version of Fortify SCA.

2.2 Initial Findings

Projects that reported no results or analysis errors were not investigated further. The following projects generated no findings:

- ECHO
- EnRUPT
- Grøstl
- MCSSHA3
- SHAvite-3
- Sarmal
- Shabal
- TIB3

The projects that did have results averaged around 37 findings.

2.3 Further investigation

When we analyzed the results a preponderance of the findings were associated with **`genKAT.c`**, an uninteresting test harness provided by NIST for purposes of the contest. Some other findings were also either uninteresting or invalid within the context.

Joy spent about 16 hours separating the findings between interesting results associated with the submitted code and the uninteresting or test harness related results, and carefully reviewed each issue.

This is in accordance with Fortify SCA's design; namely, static analysis provides automated assistance to a manual code review[7]. The reviewer spends their time auditing the critical sections of code, rather than digging around to find them (a necessary part of manual code review).

3 Results

Implementation	Buffer overflow	Out-of-bounds read	Memory leak	Null dereference
Blender	1			
CRUNCH				4
FSB			3	11
MD6	2	3		
Vortex			1	15

3.1 Blender

Category	Findings
Buffer overflow	1

Buffer overflow at Blender.c:1808

In **`Blender.c:1808`**, an apparent typographical error handles out-of-bounds memory. The length of the array, defined on line 70, is three; the highest allowable array index is 2:

```
70:    DataLength sourceDataLength2[3]; // high order parts of data length
71:           // note: the array size determines the maximum length supported
```

...

```
1802:  // deal with the length update first
1803:  bcount = ss.sourceDataLength;     // previous length
1804:  ss.sourceDataLength = bcount + databitlen;     // new length
1805:  if (ss.sourceDataLength < (bcount | databitlen))     // overflow
1806:         if (++ss.sourceDataLength2[0] == 0)            // increment higher
       order count
1807:                 if (++ss.sourceDataLength2[1] == 0)     // and the next
       higher order
1808:                         ++ss.sourceDataLength2[3];     // and the next
       one, etc.
```

Blender.c

3.2 MD6

Category	Findings
Buffer overflow	2
Out of bounds read	3

The findings comprise two instances of overwriting beyond the allocated buffer. Doubling the size of the buffer should correct multiple problems.

In addition to the buffer overflows and two related out-of-bound reads, another unrelated off-by-one problem was also found.

Buffer size issues in md6_mode.c

The MD6 implementation defined a buffer to store the final hash value in the hash state structure:

```
214: int d;           /* desired hash bit length. 1 <= d <= 512.   */
215: int hashbitlen;  /* hashbitlen is the same as d; for NIST API */
216:
217: unsigned char hashval[ (md6_c/2)*(md6_w/8) ];
218:     /* e.g. unsigned char hashval[64];  (Assumes d<=c/2.)     */
```

Defined values for md6_w (the wordsize for the algorithm, which is independent of the wordsize for the platform) and md6_c (the size of a compressed chunk) are 64 and 16, respecitively. This gives the `hashval` buffer a size of 64 bytes.

```
(md6_c/2)*(md6_w/8)
= ( 16 / 2 ) * ( 64 / 8 )
= 64 bytes.
```

This buffer size introduced four vulnerabilities:

- **Buffer overflow at md6_mode.c:611**

  ```
  610:  if (z==1) /* save final chaining value in st->hashval */
  611:     { memcpy( st->hashval, C, md6_c*(w/8) );
  612:       return MD6_SUCCESS;
  ```

  In md6_mode.c, the `memcpy()` on line 611 copies the following length:

  ```
  md6_c * ( w / 8 )
  = 16 * ( 64 / 8 )
  = 128 bytes
  ```

  into the buffer `st->hashval`, resulting in an overflow.

- **Buffer overflow at md6_mode.c:746**

```
744:  /* zero out following bytes */
745:   for ( i=full_or_partial_bytes; i<c*(w/8); i++ )
746:     st->hashval[i] = 0;
```

On line 746 of md6_mode.c, the program zeroes a 128 bytes over a 64 byte destination. The length of hashval has been shown to be 64 bytes, while `i` can increase to 127:

```
i<c*(w/8);...
i < 16*64/8
i < 128
```

(c and w are defined to be equivalent to md6_c and md6_w)

- **Out of bounds read at md6_mode.c:742**

```
740: /* move relevant bytes to the front */
741:   for ( i=0; i<full_or_partial_bytes; i++ )
742:     st->hashval[i] = st->hashval[c*(w/8)-
full_or_partial_bytes+i];
743:
```

On line 742 of md6_mode.c, the program reads from the `hashval` buffer, which has a size of 64 bytes, with an index that as large as 127. Given that `i` can be as large as `full_or_partial_bytes – 1`:

```
c*(w/8) – full_or_partial_bytes + i <=
      c*(w/8) – full_or_partial_bytes +
      full_or_partial_bytes - 1
c*(w/8) – full_or_partial_bytes + i <= c*(w/8) - 1
c*(w/8) – full_or_partial_bytes + i <= 16*64/8 - 1
c*(w/8) – full_or_partial_bytes + i <= 127
```

- **Out of bounds read at md6_mode.c:753**

```
736:   int full_or_partial_bytes = (st->d+7)/8;
```

```
748: /* shift result left by (8-bits) bit positions, per byte, if needed
*/
749:   if (bits>0)
750:     { for ( i=0; i<full_or_partial_bytes; i++ )
751:   { st->hashval[i] = (st->hashval[i] << (8-bits));
752:     if ( (i+1) < c*(w/8) )
753:       st->hashval[i] |= (st->hashval[i+1] >> bits);
```

Given that the maximum value of `st->d` is 512, `i` must be less than 64 and "`i+1`" has a maximum value of 64.:

```
i < full_or_partial_bytes
i < (st->d+7)/8
i < (512 + 7)/8
i < 64
```

The `if` statement will not prohibit this value, which will cause a read one byte beyond the end off the buffer.

All four of these issues were corrected by doubling the size of the hashval buffer in the updated MD6 implementation, made available on January 15, 2009.

Out of bounds read at md6_compress.c:280

```
421: md6_word A[5000];      /* MS VS can't handle variable size here */
```

```
279:
280: memcpy( C, A+(r-1)*c+n, c*sizeof(md6_word) ); /* output into C */
281:
```

Developer's comment: "*The read mentioned could overflow, depending on r and the size of A. If A is null, the function allocates a large enough array. At other times, the A array is declared to contain 5000 md6_words, which is large enough for all default choices of r. Ideally A's size would depend on r, but we want to statically allocate the array for performance reasons.*"

Fortify's reply: Consider returning an error code if the sizes would cause a read out of bounds. The desire to statically allocate is understandable; a bounds check is a cheap alternative way to address the issue.

3.3 CRUNCH

Category	Findings
Null dereference	4

In **crunch_224.c**:57, memory is allocated for the final block and subsequently used on line 64. This will lead to a null dereference in the event of a failure to allocate the memory. Other null dereferences stem from the unchecked allocation in the associated files **crunch_256.c**:68, **crunch_384.c**:57, and **crunch_512.c**:57.

Missing check against NULL in crunch_224.c

If the **calloc()** on line 57 fails, **mes** and then **mes_char** will be **NULL**. **mes_char** is subsequently used on line 64, exhibiting a null dereference:

```
57:    mes=(BlockType *)calloc((*nb_final_block),sizeof(BlockType));
58:    mes_char=(char*)mes;
59:    DL_char=(char*)LeftData;
60:    /*copy left data in beginning of the block*/
61:    for(i=0;i<(int)SizeLeft;i++)
62:    {
63:           temp =idx(i);
64:           mes_char[temp]=DL_char[temp];
```

3.4 FSB

Category	Findings
Memory leak	2
Null dereference	11

Memory leak in fsb.c:209

Fortify found three memory leaks in **fsb.c**, in **Hash** and **HashFile**. In **fsb.c**:204 the function allocates memory for the state variable. If **Init** or **Update** do not return **SUCCESS**, the function returns without freeing **hashState**. Similar memory leaks exist for **state** (**fsb.c**:222) and **buffer** (**fsb.c**:223) in the **HashFile** function:

```
204:   HashReturn Hash(int hashbitlen, const BitSequence *data, DataLength
       databitlen, BitSequence *hashval) {
205:          hashState* state = (hashState*) malloc(sizeof(hashState));
206:          int return_value;
207:          return_value = Init(state, hashbitlen);
208:          if (return_value != SUCCESS) {
209:            return return_value;
210:          }
```

Missing check against NULL in fsb.c:53

Another issue throughout **fsb.c** is a failure to check for the success of memory allocations. If **malloc** (or similar functions) are unable to successfully allocate memory, the return value is NULL. At 11 points in the file, memory allocations are used without checking the return value, risking a null dereference. The unchecked allocations occur on lines 51, 53, 54, 65, 155, 164, 175, 176, 205, 222, and 223.

In the following example, **state->first_line** may be set to NULL on line 51. It is then used on line 53:

```
50:    /* compute the first QC matrix line */
51:      state->first_line = (unsigned char***) malloc(state->b*sizeof(unsigned
       char**));
52:      for (k=0; k<state->b; k++) {
53:    state->first_line[k] = (unsigned char**) malloc(8*sizeof(unsigned
       char*));
```

3.5 Vortex

Category	Findings
Memory leak	1
Null dereference	15

Fortify found two security sensitive issues, one with multiple instances.

An issue that occurs throughout **SHA3api_ref.c** and **vortex_core.c** is a failure to check memory allocations. If **malloc**, or similar functions, are unable to allocate memory, they will return null. At 15 points in the file, memory allocations are used without checking against null, which could cause a null dereference. The allocations in question occur in **SHA3api_ref.c** (111, 112, 113, 120, 121, 122, 157, 173, 282) and **vortex_core.c** (739, 740, 741, 742, 793, 794).

Missing check against NULL in SHA3api_ref.c

In the following example, **state->hash** could be set to NULL on line 111 and then dereferenced on line 114:

```
107:   switch(hashbitlen)
108:   {
109:         case 224:
110:         case 256:
111:               state->hash = (uint8_t *)malloc(32);
112:               state->a0_b0 = (uint8_t *)malloc(32);
113:               state->ta_tb = (uint8_t *)malloc(32);
114:               varcpy(state->hash,  a0_b0_32_g, 32);
```

Memory leak in SHA3api_ref.c:299

Fortify also found a memory leak in the Hash function. **SHA3api_ref.c:282** allocates memory for variable **buf**. If an invalid hash length has been provided, the function will return at line 299 without freeing the memory:

```
282:   buf = (BitSequence *)malloc((databitlen+7)/8);
283:   varcpy(buf, (uint8_t *)data, (uint32_t)((databitlen+7)/8));
284:   set_format((BitSequence *)buf, databitlen);
285:   switch(hashbitlen)
286:   {
287:         case 224:
288:         case 256:
289:               error_code = tunable_vortex(buf, l, (uint8_t
       *)hashval, (uint32_t)hashbitlen,
290:                     number_of_rounds_g, mul_type_g, a0_b0_32_g,
       ta_tb_32_g, degree_of_diffusion_g);
291:               break;
292:         case 384:
293:
294:
295:                     number_of_rounds_g, mul_type_g, a0_b0_64_g,
       ta_tb_64_g, degree_of_diffusion_g);
296:               break;
297:         default:
298:               perror("Hash(): bad hash type\n");
299:               return BAD_HASHBITLEN;
300:   }
```

4 Appendix

4.1 Concessions

The following projects were not downloaded from the NIST site because they were marked "Submitter has conceded that the algorithm is broken" at the time the projects were downloaded:

- Abacus
- BOOLE
- DCH
- Khichidi-1
- MeshHash
- StreamHash
- Tangle
- WaMM
- Waterfall

The authors of the following conceded after the projects were downloaded and while the analysis was already underway:

- SHAMATA

4.2 Endnotes

[1] http://tech.slashdot.org/article.pl?sid=08/12/21/1334238&tid=93

[3] In 1999, a bug in the RSA reference implementation was responsible for vulnerabilities in OpenSSL and two separate SSH implementations (see http://www.cert.org/advisories/CA-1999-15.html).

[4] http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/submissions_rnd1.html

[5] On February 12, there were 43 projects that were not yet conceded (see "Concessions" in the appendices).

[6] http://csrc.nist.gov/groups/ST/hash/sha-3/documents/KAT1.zip

[7] Chapter 3, "Static Analysis as Part of the Code Review Process," p. 47, *Secure Programming with Static Analysis*, Chess and West, 2007.