14:09 Postscript that shows its own MD5

by Gregor “Greg” Kopf

Introduction

Playing with file formats to produce unexpected results has been a hacker past-time for quite a while. These odd results often include self-referencing code or data structures, such as zip bombs, self-hosting compilers, or programs that print their own source code—called quines. Quines are often posed as brain teasers for people learning new programming languages.

In the light of recent attacks on the cryptographic hash functions MD5 and SHA-1, it is natural to ask a related question: Is there a program that prints out its own MD5 or SHA-1 hash? A similar question has been posed on Twitter by Melissa.30

The original tweet is from 2013. It appears that since then nobody provided a convincing solution because in March 2017 Ange Albertini declared that the challenge was still open. This brought the problem to my attention—the perfect little Sunday morning challenge.

A Bit of Context

Melissa’s challenge asks whether there is a document in a conventional format (such as PDF) which mentions its own MD5 or SHA1 hash in the text and is right.

While this method can likely be applied to Postscript documents as well, I did not like it very much. Computing the MD5 hash of the program at runtime felt like cheating.

The desired file is a modified fixpoint of the used hash function, in the same sense that this program is a modified quine. A plain fixpoint would be a value \( x \) where \( x = h(x) \). Here, \( h \) denotes the hash function. This problem has not yet, so far as I know, been solved constructively. (Statistics reveals that such fixpoints exist with a certain probability, however.)

```python
a=['"from hashlib import *", '+n=n.join(a)+n)
print md5("a="+str(a)+n+n.join(a)+n).hexdigest()
from hashlib import *
n=chr(10)
print md5("a="+str(a)+n+n.join(a)+n).hexdigest()
```

The problem of building such a program can be viewed from (at least) two different angles. One could view this hypothetical program as a modified quine: instead of printing its own source code, the program prints the hash of its own source code. If you are familiar with how quines can be generated, you can easily see that the following program is indeed a solution to the question:

---

30https://twitter.com/0xabad1dea/status/365863999520251906
Fortunately, we are looking for something a little easier. We are looking for an x that satisfies $x = \text{encode}(h(x))$ for some encoding function encode(). I decided to chase this idea: constructing such a value x, using MD5 as hash function h() and a function that builds a Postscript file as encode().

But there is another— even simpler— trick one can perform given the ability to create colliding MD5 inputs. One can create two executables with the same MD5 hash but with different semantics. The general idea is to generate two colliding MD5 inputs a and b. We can then write a program like the following.

```python
print 'Hi, my message is:'
if a == b:
    print "Hello World"
else:
    print "Oh noez, I've been hacked!!!"
```

And another program like this:

```python
print 'Hi, my message is:'
if b == b:
    print "Hello World"
else:
    print "Oh noez, I've been hacked!!!"
```

Both programs will have the same MD5 hash; in the second program, we only replaced a with b.

But why does this work? There are two things one needs to pay attention to. Firstly, we have to understand that while the inputs a and b might collide under MD5, the strings "foo" + a and "foo" + b may not necessarily collide. Fortunately, as MD5 is a Merkle–Damgård hash, given two colliding inputs a and b, the hashes \( \text{MD5}(a + x) \) and \( \text{MD5}(b + x) \) will also collide for all strings x. This property allows us to append arbitrary content after the colliding blocks.

The Basics

When Wang et al., broke MD5 in 2005, there was considerable interest in what one could do with a chosen-prefix MD5 collision attack. Sotirov et al., have demonstrated in 2008 that one could exploit Wang's work in order to build a rogue X.509 CA certificate—the final nail in MD5's coffin.
Constructing the Target

Using the above technique allows us to encode a single bit of information into a program without changing the program’s MD5 hash. Can we also encode more than one bit into such a program? Unsurprisingly, we can!

We start the same way that we have already seen, by generating two MD5 collisions \( a \) and \( b \) that share the following prefix.

```python
print 'Hey, I can encode multiple bits!'
result = []
if a == b:
    result.append(0)
else:
    result.append(1)
```

This allows us to build two colliding programs that look like the following. (Exchange \( a \) with \( b \) to get the second program.)

```python
print 'Hey, I can encode multiple bits!'
result = []
if a == b:
    result.append(0)
else:
    result.append(1)
if c == d:
    result.append(0)
else:
    result.append(1)
```

And from here, we simply iterate the process, computing two colliding MD5 inputs \( c \) and \( d \) that share this prefix.

```python
print 'Hey, I can encode multiple bits!'
result = []
if a == b:
    result.append(0)
else:
    result.append(1)
if c == d:
    result.append(0)
else:
    result.append(1)
```

This allows us to build a program with two bits that might be adjusted without changing the hash.

```python
print 'Hey, I can encode multiple bits!'
result = []
if a == b:
    result.append(0)
else:
    result.append(1)
if c == d:
    result.append(0)
else:
    result.append(1)
```

We can replace \( a \) with \( b \), and we can replace \( c \) with \( d \). In total, this yields four different programs with the same MD5 hash. If we add a statement like `print result` at the end of each program, we have four programs that output four different bit-strings but share a common MD5 hash!

How does this enable us to generate a program that outputs its own MD5 hash? We first generate a program that we can encode 128 bits into. Knowing that the MD5 hash of this program will not change independently from what bits we encode into the program. Therefore, we simply encode the 128 output bits of MD5 into the program without altering its hash value. In other words, the program prints the 128 output bits of its own hash value.

Application to Postscript

This technique can directly be applied to Postscript documents as Postscript is a simple, stack-based language. Please consider the following code snippet.

```postscript
(a)
(b)
eq
{1
0
}ifelse
```

NEW! DS-CAPS

$89.00*

A Unique Keyboard or Program Activated Data Switch for the IBM PC or Any MS-DOS System.

This compact self-powered switch is software controlled at the keyboard or program to direct parallel data from the computer to printers/plotters. Eliminates the time and frustration of retyping data from one peripheral to another. To install, connect between computer and peripherals, plug into power outlet and posterior, plug into power outlet and you are ready to code select and direct data flow between the two devices.

Also available is a complete line of manual and electronic switches plus converters to interface and direct data between CPUs and peripherals.

VIA WEST, Inc.
The Interface Company
53 North Stone Ave. Tucson, AZ 85705
(520) 623-5776

©1983-1985 ViaWest Inc.
All rights reserved. Printed in U.S.A.
While this may look a bit cryptic, the program is in fact very simple. It compares the string literal “a” to the string literal “b”, and if both strings are equal, it pushes the numeric value 1 to the stack. Otherwise, it pushes a 0.

This example highlights the manner in which we can build a Postscript file that we encode 128 bits of information into without changing the file’s MD5 hash. The program will push these desired bits to the stack. We can extend this program with a routine that pops 128 bits off the stack and encodes them in hex. To demonstrate the feasibility of this idea, we can inspect how one nibble of data would be handled by this routine.

```
0 eq
  { 0 eq
    { 0 eq
      { (0)
        }
        (1)
        ifelse
      }
      0 eq
        { (2)
          }
          (3)
          ifelse
        }
        ifelse
    }
    ifelse
  }
  ifelse

... show
```

This code excerpt will pop four bits off the stack. If all bits are zero, the string literal “0” will be pushed onto the stack. If the lowest bit is a one and all other bits are zero, the string literal “1” will be pushed, etc. The `show` statement at the end causes the nibble to be popped off the stack and written to the current page.

An example of such a Postscript document is included in the feelies. If you want to build such a document on your own, you could use the python-md5-collision library to build MD5 collisions with chosen prefixes.

```
$ md5sum poc.ps
768d9d89d2bc825a319eb8962ad30580  poc.ps
```

Closing Remarks

We have seen two approaches for generating programs that print out their own hash values. The quine approach does not require a collision in the used hash function, however this comes at the cost of language complexity. In order to build such a modified quine, the chosen language must allow for self-referencing code as well as computing the selected hash function.

The fixpoint approach is computationally more expensive to implement, as several hash collisions must be computed. However, these hash calculations can be performed in any programming environment. With this approach, the target language can be comparably simple: it just needs conditionals, string comparison and some method to output the result.

31 unzip pocorgtfo14.pdf md5.ps
32 git clone https://github.com/thereal1024/python-md5-collision