Managing Risk and Security since 1998

Crypto Pet Peeves: Hashing…Encoding…It's All The Same, Right?

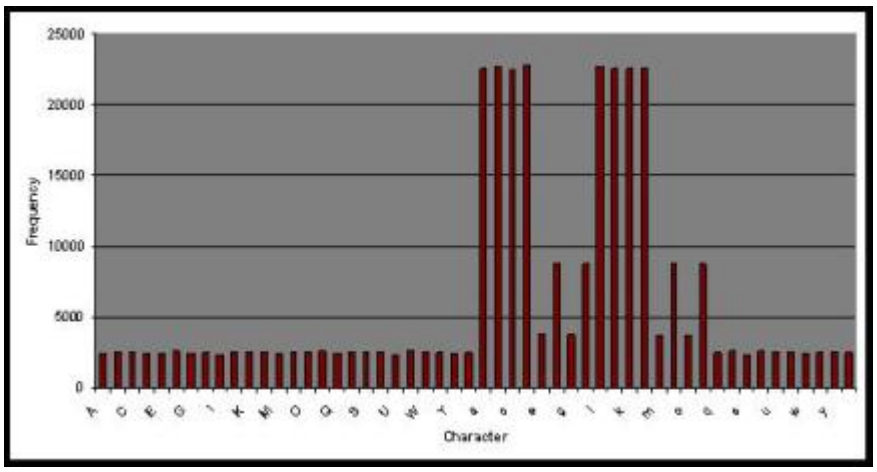
Patrick Toomey

We all know cryptography is hard. Time and time again we in the security community give advice that goes something like, "Unless you have an unbelievably good reason for developing your own cryptography, don't!". Even if you think you have an unbelievably good reason I would still take pause and make sure there is no other alternative. Nearly every aspect of cryptography is painstakingly difficult: developing new crypto primitives is hard, correctly implementing them is nearly just as hard, and even using existing crypto APIs can be fraught with subtlety. As discussed in a prior post, Seed Racing (http://labs.neohapsis.com/2008/04/29/seed-racing/), even fairly simple random number generation is prone to developer error. Whenever I audit source I keep my eyes open for unfamiliar crypto code. So was the case on a recent engagement; I found myself reviewing an application in a language that I was less familiar with: Progress ABL.

Progress ABL is similar to a number of other 4GL languages, simplifying development given the proper problem set. Most notably, Progress ABL allows for rapid development of typical business CRUD applications, as the language has a number of features that make database interactions fairly transparent. For those of you interested to learn more, the language reference manual (http://www.psdn.com/library/servlet/KbServlet/download/4819-102-14253/dvref.pdf) can be found on Progress' website.

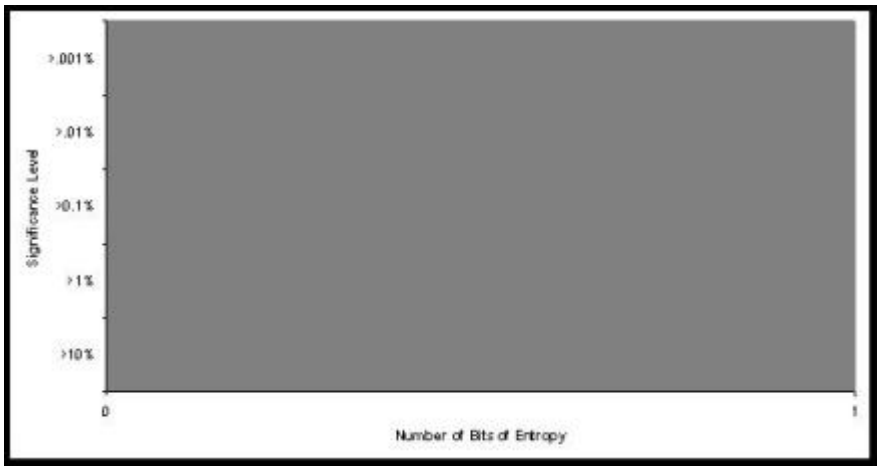
As I began my review of the application I found myself starting where I usually do: staring at the login page. The application was a fairly standard web app that required authentication via login credentials before accessing the sensitive components of the application. Being relatively unfamiliar with ABL, I was curious how they would handle session management. Sure enough, just as with many other web apps, the application set a secure cookie that uniquely identifies my session upon login. However, I noticed that the session ID was relatively short (sixteen lower/upper case letters and four digits). I decided to pull down a few thousand of the tokens to see if I noticed any anomalies. The first thing I noticed was that the four digit number on the end was obviously not random, as values tended to repeat, cluster together, etc. So, the security of the session ID must lie in the sixteen characters that precede the four digits. However, even the sixteen characters did not look so random. Certain letters appeared to occur more than others. Certain characters seemed to follow other characters more than others. But, this was totally unscientific; strange patterns can be found in any small sample of data. So, I decided to do a bit more scientific investigation into what was going on.

Just to confirm my suspicions I coded up a quick python script to pull down a few thousand tokens and count the frequency of each character in the token. Several minutes later I had a nice graph in excel.



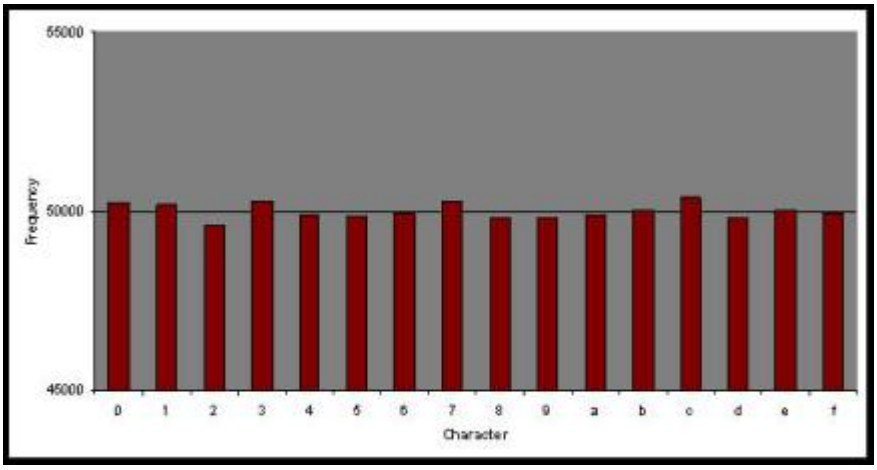
(http://neolab.files.wordpress.com/2008/08/distribution_encode.jpg)
Histogram of Encode Character Frequency

Ouch! That sure doesn't look very random. So, I opened up Burp Proxy and used their Sequencer to pull down a few thousand more session cookies. The Burp Sequencer has support for running a number of tests, including a set of FIPS-compliant statistical tests for randomness. To obtain a statistically significant result Burp analyzes a sample size of 20,000 tokens. Since I saw that the four digit token at the end of the session ID provided little to no entropy, I discarded them from the analysis. It seemed obvious that the sixteen character sequence was generated using some sort of cryptographic hash, and the four digit number was generated in some other way. I was more interested in the entropy provided by the hash. So, after twenty minutes of downloading tokens, I let Burp crunch the numbers. About 25 seconds later Burp returned an entropy value of 0 bits. Burp returned a graph that looked like the one below, showing the entropy of the data at various significance levels.

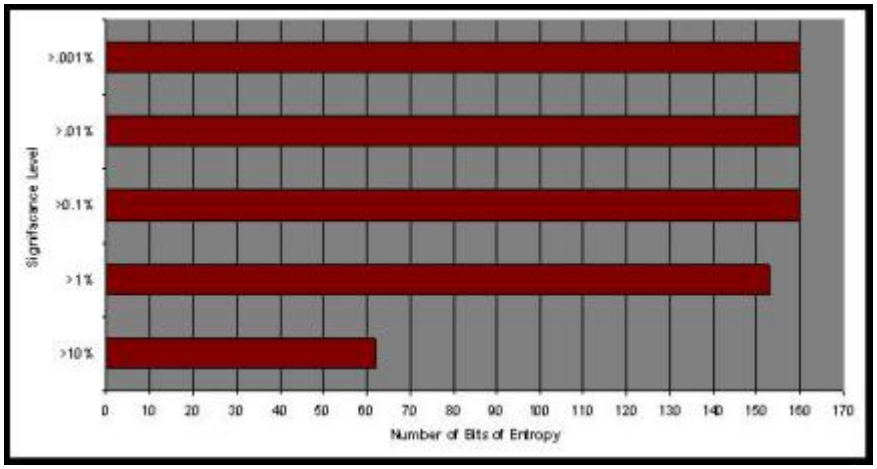


(http://neolab.files.wordpress.com/2008/08/statistical_entropy_encode.jpg)
Encode Entropy Estimation

Hmmm, maybe Burp is broken. I was pretty sure I had successfully used the Burp Sequencer before. Maybe it was user error, a bug in the current version, who knows. I decided that a control was needed, just to ensure that the tool was working the way I thought it should. So, I wrote a bit more python to simply print the hex-encoded value of a SHA1 hash on the numbers 1-20,000. I loaded this data into Burp and analyzed the data set. Burp estimated the entropy at 153 bits. Just to compare with the prior results, here is the distribution graph and the Burp entropy results for the SHA1 output:



(http://neolab.files.wordpress.com/2008/08/distribution_sha1.jpg)
Histogram of SHA1 Character Frequency



(http://neolab.files.wordpress.com/2008/08/statistical_entropy_sha1.jpg)
SHA1 Entropy Estimation

I repeated the same test against a set of JSESSIONID tokens and found a similarly acceptable result. Ok, so the Burp Sequencer seems to be working.

So, I next went hunting for the session token generation code in the application. After a little greping I found the function for generating new session tokens. Ultimately the function took a number of values and ran them through a function called "ENCODE". Hmmm, ENCODE, that didn't sound familiar. Some more greping through the source did not reveal any function definitions, so I assumed the function must be part of the standard library for ABL. Sure enough, on page 480 of the language reference manual there was a description of the ENCODE function.

"Encodes a source character string and returns the encoded character string result"

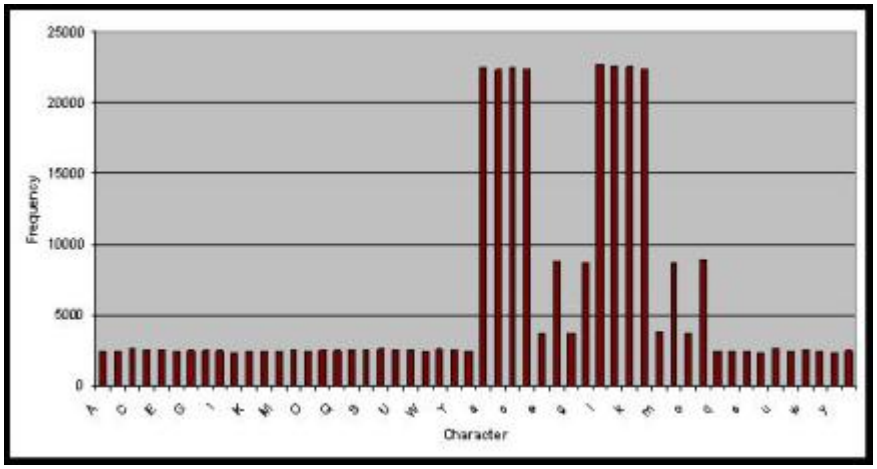
The documentation then goes on to state:

"The ENCODE function performs a one-way encoding operation that you cannot reverse. It is useful for storing scrambled copies of passwords in a database. It is impossible to determine the original password by examining the database. However, a procedure can prompt a user for a password, encode it, and compare the result with the stored, encoded password to determine if the user supplied the correct password."

That is the least committal description of a hash function I've ever had the pleasure reading. It turns out

the application, as well as a third party library the application depends upon, uses this function for generating session tokens, storing passwords, and generating encryption keys. For the sake of reproducibility I wanted to be sure my data was not the result of some strange artifact in their environment. I installed the ABL runtime locally and coded up a simple ABL script to call ENCODE on the numbers 1-20000. I reran the Burp Sequencer and got the exact same result, 0 bits.

At this point I was fairly sure that ENCODE was flawed from a hashing perspective. A good quality secure hash function, regardless of how correlated the inputs are (as the number 1-20000 obviously would be), should produce output that is indistinguishable from truly random values (see Cryptographic Hash Functions (http://en.wikipedia.org/wiki/Cryptographic_hash_function) and Random Oracle Model (http://en.wikipedia.org/wiki/Random_oracle) for more information). ENCODE clearly does not meet this definition of a secure hash function. But, 0 bits, that seems almost inconceivably flawed. So, giving them the benefit of the doubt, I wondered if the result is dependent on the input. In other words, I conjectured that ENCODE might perform some unsophisticated "scrambling" operation on the input, and thus input with low entropy will have low entropy on the output. Conversely, input with high entropy might retain it's entropy on output. This still wouldn't excuse the final result, but I was curious none the less. My final test was to use the output of my SHA1 results and feed them each through the ENCODE function. Since the output of the SHA1 function contains high entropy I conjectured that ENCODE, despite its obvious flaws, might retain this entropy. The results are shown below:



(http://neolab.files.wordpress.com/2008/08/distribution_sha1_encode.jpg)
Histogram of SHA1 then Encode Character Frequency

(http://neolab.files.wordpress.com/2008/08/statistical_entropy_sha1_encode.jpg)
SHA1 then Encode Entropy Estimation

ENCODE manages to transform an input with approximately 160 bits of entropy into an output that, statistically speaking, contains 0 bits of entropy. In fact, the frequency distribution of the character output is nearly identical to the first graph in this post.

This brings me back to my opening statement, "Unless you have an unbelievably good reason for developing your own cryptography, don't!". I can't figure out why this ENCODE function exists? Surely the ABL library has support for a proper hash function like SHA1, right? Yes, in fact it does. The best explanation I could come up with is that it is a legacy API call. If that is the case then the call should be deprecated and/or documented as suitable only in cases where security is of no importance. The current API does the exact opposite, encouraging developers to use the function for storing passwords. Cryptography is hard, even for those of us that understand the subtlety involved. Anything that blurs the line between safe and unsafe behavior only makes the burden on developers even greater.

It is unclear, based on this analysis, how much effort it would require to find collisions in ABL's ENCODE function. But, even this simple statistical analysis should be enough for anyone to steer clear of its use for anything security related. If you are an ABL developer I would recommend that you try replacing ENCODE with something else. As a trivial example, you could try: HEX-ENCODE(SHA1-DIGEST(input)). Obviously you need to test and refactor any code that this breaks. But, you can at least be assured that SHA1 is relatively secure from a hashing perspective. That said, you might want to start looking at SHA-256 or SHA-512, given the recent chinks in the armor of SHA1:

- SHA-1 Broken (http://www.schneier.com/blog/archives/2005/02/sha1_broken.html)
- Notes on the Wang et al. 2^63 SHA-1 Differential Path (http://eprint.iacr.org/2007/474)

Unfortunately, it does not appear that ABL has support for these more contemporary SHA functions in their current release.

Ok….slowly stepping down off my soapbox now. Bad crypto just happens to be one of my pet peeves.

Footnote:

Just before posting this blog entry I decided to email Progress to see if they were aware of the behavior of the ENCODE function. After a bit a few back and forth emails I eventually got an email that desribed the ENCODE function as using a CRC-16 to generate it's output (it is not the direct output, but CRC-16 is the basic primitive used to derive the output). Unfortunately, CRCs were never meant to have any security gurantees. CRCs do an excellent job of detecting accidental bit errors in a noisy transmission medium. However, they provide no gurantees if a malicous user tries to find a collision. In fact, maliciously generating inputs that produce identical CRC outputs is fairly trivial. As an example, the linearity of the CRC-32 alogirthm was noted as problematic in an analysis of WEP (http://www.isaac.cs.berkeley.edu/isaac/wep-faq.html). Thus, despite the API doc recommendation, I would highly recommend that you not use ENCODE as a means of securely storing your user's passwords.

This entry was posted on Monday, August 25th, 2008 at 7:29 pm and is filed under Information Security. You can follow any responses to this entry through the RSS 2.0 feed. You can leave a response, or trackback from your own site.

8 Responses to *Crypto Pet Peeves: Hashing…Encoding…It's All The*

Same, Right?

The Grumpy Hacker says:
August 29, 2008 at 4:57 pm
Nice work.

Reply
Sam says:
September 18, 2008 at 1:57 pm
hashing…

I can't believe I missed this! I'm going to have to do some more reading me thinks….

Reply
Recent Links Tagged With "artifact" - JabberTags says:
October 1, 2008 at 11:49 am
[...] public links >> artifact Crypto Pet Peeves: Hashing…Encoding…It's All The Same, Right? Saved by hebahamdy on Mon 29-9-2008 Squeezing Down syndrome out of our culture Saved by [...]

Reply
Peter Gien says:
October 3, 2008 at 8:41 pm
Great article and excellent sleuthing work. You're quite right for taking them to task for using CRC to hash passwords. In fact I can think of no reason to use this ENCODE function to do anything useful at all.

Reply
Nunes says:
July 13, 2010 at 5:41 pm
Do you know the algorithm ENCODE() utilized? Or any way to discover it?

Thanks!

Reply
Rob says:
September 23, 2010 at 9:52 am
Excellent finding! I found your blog while looking for information on how the encode function is implemented.

Thanks!

Reply
Gopal says:
November 16, 2011 at 9:23 am
Great article! The way you analyzed is excellent.

Reply
Greta (@greatskuldy) says:
March 14, 2012 at 7:45 pm

I was looking information about sha1 in progress ABL and I found this! woow excellent work! I would have your article in mind when I build my web services.

Reply