Homomorphic Encryption

Current best practices for both HIPAA and the PCI are encryption of “data in motion” and “data at rest”. If you were around earlier, we discussed the protection of data in motion during my morning talk. If you missed it, you can download a copy of those slides, as well as these, from my company’s web site, Formularity.com. This talk will focus in on the problems of the encryption and protection of data at rest. This actually doesn’t seem to be that big a deal, right? We just stuff all the data we have collected from our customers, citizens, or patients into an encrypted database. After all, we have decades of experience with relational databases, and now the new family of No-SQL databases like Mongo and CouchDB. What do we use databases for? Well, databases store information in a form that allows for selective retrieval and manipulation. If we weren’t going to retrieve our stored data selectively, why bother using an expensive, complex, database? Why not just put all our data into one big text file? Well, we do want to selectively retrieve our data. Quite often, we want to “rapidly” retrieve our data. To that end, databases have indices. We can use indices to pick all the records that were before a certain date, which are between two values, or which contain certain keywords. When we retrieve the data, we may wanted it sorted and organized in a specific manner. Alphabetized, or grouped by Sales Region. This is all accelerated by indices. We put the data into the database and it both stores the data and builds indices about it. Then it waits for our requests. We send it retrieval requests, along with the parameters that describe what we are looking for. The database compares our parameters against the information it has in both the records and the indices and returns to us the results, formatted as we requested. All fine and dandy.

Now, let’s consider how this changes with encryption. First of all, we have to supply unencrypted data to the database so it can properly index it. The database then encrypts both our data and its indices, since indices often hold pieces of our information. But now, the use of the indices is restricted. You can do a full match on the index, but you can’t do ranges or partial matches. Most databases that support encryption allow you to do joins on tables, but not foreign key constraints. Given the severe limitations on indexing encrypted data, most database architects forgo complete record encryption and just encrypt the record elements that are thought to be sensitive, like credit card number or SSN. It’s a compromise. You protect some of the information and can still derive some of the benefits of the database. Not a perfect solution, considering that some or most of our collected information is in the clear. The type of data that we are often concerned about is personally identifiable information (PII). PII are the attributes that fairly uniquely describe a particular person. As I said, SSN is PII. Hopefully, there are no two in the world with the same SSN. A name is PII, but only when it’s associated with other characteristics that are semi-unique to the person. There are plenty of Brad Whiteheads out there. However, the number of Brad Whiteheads out there with my birthdate are hopefully small. Add in where I was born and chances are you’ve found a combination that’s unique to me.

Determining what data, aggregated together, is considered PII is not trivial. The US Federal government and other governmental agencies have come up with detailed charts that define exactly what combination of which characteristics are officially defined as being PII. And these change! So, the fields you decide to leave as plain text today may become PII in the future. But privacy goes beyond just PII. I’d say it’s anything you wouldn’t want publicly disclosed. It’s your medical treatments. Perhaps it’s

even your medical queries to Google. You do know that Google has experimented with tracking medical inquiries trends in the US and making them available to the Federal Government don't you? It was part of a program to detect outbreaks of illnesses, in conjunction with doctor and clinic reports. If a lot of people in San Diego start asking Google about symptoms involving high fevers, stomach irritations, and weakness, there may be a flu epidemic in progress. But that's not the direct subject of our talk today. Again, let's just say that sensitive information is anything you don't want other people knowing. If you have an account on Ashley Madison, you might not even want people to know your name. Personally, I don't want anybody to know I'm a member of the Taylor Swift Official Fan Club – it ruins my macho image ☺.

Another alternative to encrypting the database is to encrypt the storage on which the database rests. This is like whole disk encryption on your computer. Like Microsoft's Bit Locker or the freely available TrueCrypt program. The database itself isn't encrypted. It reads and writes into encrypted disk space. So, no changes are required to how the database operates. However, this type of encryption is really only effective against somebody stealing the database, not against accessing and stealing the contents of the database. It also doesn't do anything for protecting the data coming into the database, the queries against the database, and the output of the database. Still, disk encryption is better than nothing!

Column and disk encryption might be considered acceptable for a database in a corporate data center. The assumption is that the corporation will restrict access to the database to only those people that require it and are approved to deal with its contents. After all, if it's a bank database, you expect a certain type of banker to have access to the information, even if it's PII or sensitive. Likewise, you expect that the IT staff running the data center to have been vetted, to be trustworthy, and for proper auditing controls to be in place. The IT staff in particular have access to the crypto keys used to encrypt the columns and or the disks.

Now, let's turn our attention to The Cloud. In this context, I'm referring to commercial data centers that provide computing and storage services to the public. This form of The Cloud is often referred to as "Infrastructure as a Service (IaaS)", or "Platform as a Service (PaaS)". Examples of this type of Cloud are Amazon Web Services, Google Cloud Platform, Microsoft Azure, and RackSpace, just to name a very few of the largest and best known names. This morning I addressed why The Cloud is a good thing from a data processing standpoint. Basically these cloud providers allow companies to account for budget and processing fluctuations, as well as allowing businesses to concentrate of their core competencies and not data processing.

So, your shiny new eCommerce website or medical app backend is "in the cloud", running on Amazon Web Services or Google Compute Engine. You are collecting customer or patient account information. Where are you storing it? First off, unless you are using client-side encryption, you have already exposed all your customer and patient data as plain text to the cloud provider when your web server

received it. That was the subject of this morning's talk. If you truly want to protect your customer and patient information, and your company's financial future, download this morning's slides and investigate client-side encryption.

You put the collected data in a database in the cloud. But we've already addressed how the IT staff has access to the decryption keys. And this IT staff aren't your employees. You don't know who they are. You are trusting Google or Microsoft or RackSpace. Look how well this worked for the FBI, the NSA and the CIA. These are intelligence community organizations that routinely subject their employees to surveillance and polygraph testing. Yet, both Aldrin Ames (CIA) and Robert Hanssen (FBI), sold information to the Russian intelligence service. Chelsea Manning, US Army Intelligence, collected and turned over 250,000 classified and sensitive documents to Wiki Leaks. Edward Snowden, a NSA contractor, stole perhaps as many as 1.7 million top secret and sensitive documents from the US DOD, the British GCHQ, and the Australian Intelligence service.

This person is a contractor, working for your cloud provider. (Still) Just kidding!!! If your customer records are stolen out of the cloud data center, it won't be RackSpace paying for the years' worth of credit monitoring for each exposed customer. Again, please don't get me wrong. The vast majority of system administrators are dedicated, trustworthy individuals. Still, Carnegie Mellon University's Software Engineering Institute has an Insider Threat team that, as of 2014, had a database of over documented 1000 insider thief incidents, at both corporate and cloud data centers.

So what's the solution? Wouldn't it be nice if we could just encrypt our customer and patient information as we collect it and send it into a database, still encrypted? And, while we are wishing and dreaming, wouldn't it also be nice if we could send encrypted queries to our cloud database? Finally, it would be great of our database could send back the results, encrypted so that only we could see them? All of this without sharing our crypto keys with the cloud provider and having to trust them?

Well we can! The solution is called homomorphic encryption. Basically, homomorphic encryption is a form of encryption that allows operations to be performed on encrypted information without having to decrypt it. In the case of databases, it's possible to use homomorphic encryption to store, query, retrieve, and return database information without ever decrypting it! Since we've seen the difficulties involved in using conventional encryption with a database, homomorphic encryption sounds too good to be true. But it is!

OK, let's drop down a level and explore a bit about how homomorphic encryption works. First, we have known about the possibility of homomorphic encryption as long as we've known about RSA-type public key encryption. Ron Rivest, the 'R' in RSA, observed in 1979 that regular RSA public key encryption used without padding exhibited a type of homomorphism. If you encrypt two integers with the same public key and multiply their ciphertext, when you decrypt the product with the private key, you will get the product of the original two integers. So, RSA PKI is homomorphic in regards to multiplication. Rivest predicted that there were forms of encryption that would by homomorphic in regards to all operations. This kicked off the search. In 1999 Pascal Paillier developed an encryption that is homomorphic in

regards to addition. As simple as that may sound, Paillier encryption opens the door to true democracy! Voters can encrypt their votes using Paillier encryption and submit the ciphertext for aggregation. All the encrypted votes can be summed without being decrypted. The final tally is then decrypted and the winner revealed. Before somebody says – but won't all the 'yea' votes encrypt to the same value? How is that a secret ballot? The answer is that Paillier encryption includes a random factor, making each vote's ciphertext unique, while still decrypting to the original value.

RSA and Paillier encryption are known as Somewhat Homomorphic Encryption (SHE). Despite Rivest's prediction, nobody was sure that Fully Homomorphic Encryption (FHE) was possible. The real breakthrough came in 2009 when Craig Gentry, in his PhD thesis, described a semi-practical fully homomorphic solution. I say semi-practical because Gentry's method requires a 3GByte key size and a database search would run 1 trillion times slower than its plaintext counterpart. Still, Gentry proved that full homomorphic encryption is possible. That has opened the floodgates for both research and funding. BTW, do we have any electrical or communications engineers here? Gentry's insight into the problem was truly phenomenal. He treated the problem as one of noise in a communications circuit, and like in communications, he added error-correcting coding, much like Reed-Solomon encoding. He also used signal regeneration to re-baseline the "signal".

Once the possibility of true homomorphic encryption was proven, a number of optimizations and specializations to both SHE and FHE were developed. Since Gentry's breakthrough, IBM, where Gentry now works, has released an open source building block library – HELib, to help advance research. Researchers at a number of universities have in turn built on HELib to develop truly practical homomorphic encryption products. I'll demonstrate a homomorphic encryption database today that was developed at MIT. We don't have practical FHE yet, but various SHE schemes have been perfected and combined such that the MIT research team found that 99.5% of all real-world database queries can be satisfied by their homomorphic encryption database, with an overhead of only 14% to 26%. I'll trade 25% reduction in speed for the ability to use cloud resources with no risk of data loss or compromise.

Not only is the data in the database always encrypted, but the queries sent to the database are also encrypted. Nobody outside of the requester can ever see data, queries, or results. For small results, it's not even possible to detect if the query returned a result at all.

So, now you can see why, if you haven't heard of homomorphic encryption before this, you'll be hearing a lot about it in the next several years. All of the large commercial database vendors are preparing to release homomorphic versions of their flagship databases. You can already use the MIT homomorphic encryption in Google's Encrypted Big Query, a version of their Big Query semi-SQL database. SAP has a homomorphic encryption front-end for their Hana in-memory database. Microsoft has announced Always Encrypted SQL Server as part of their SQL Server 2016 family of products. Since Craig Gentry is part of a research team at IBM, I think it's safe to assume that there will be a homomorphic version of

DB2. My own company is using homomorphic encryption to help our client's protect their customer's data, while giving them the flexibility to use the cloud.

The MIT database I'll demonstrate might seem a bit unique at first, but if you think about it, it will make perfect sense. It's not actually a database itself. It uses MySQL as its database. It's a proxy that sits between a client and the MySQL server. As such, it encrypts all the data before it's sent to the MySQL Server for storage. When the client issues a normal SQL query, the proxy restructures the query and encrypts the query parameters so MySQL's normal query processor will work correctly. The MySQL server returns encrypted results – after all, it never saw any plain text at all – in either the data or the query. The proxy decrypts the results and gives the plain text back to the client. The whole operation is transparent to both the client and the server. Only the proxy ever has the crypto keys, and by running the proxy on the client, not the server, you can guarantee that the server (and its IT staff) never have the keys. The homomorphic encryption is so transparent that existing applications can work with the proxy without any changes in their coding. It seems that MySQL is a very popular database at MIT. In fact, MIT runs several large MySQL server instances to power some of their academic applications, as well as a number of WordPress instances. The MIT research team convinced a several of these MySQL users to move from one of the existing MySQL server instances over to their homomorphic proxy version. No changes to WordPress were required. Several minor annotations were required of the SQL schema to support the academic applications. For certain queries, you have to give the proxy hints about how the data should be stored and queried.

Before we get into the demonstration, let me say that between the client-side encryption I described this morning, and homomorphic databases, we now have the tools to keep our customer, citizen, and patient data truly private and secure.

<DEMO>