Understanding and Selecting a Tokenization Solution

Author's Note

The content in this report was developed independently of any sponsors. It is based on material originally posted on the Securosis blog but has been enhanced, reviewed, and professionally edited.

Special thanks to Chris Pepper for editing and content support.

Licensed by RSA

RSA, The Security Division of EMC, is the premier provider of security solutions for business acceleration, helping the world's leading organizations succeed by solving their most complex and sensitive security challenges. RSA's information-centric approach to security guards the integrity and confidentiality of information throughout its lifecycle - no matter where it moves, who accesses it or how it is used. RSA offers industry-leading solutions in identity assurance & access control, data loss prevention, tokenization, encryption & key management, compliance & security information management and fraud protection. These solutions bring trust to millions of user identities, the transactions that they perform, and the data that is generated. For more information, please visit www.RSA.com and www.EMC.com.

Contributors

The following individuals contributed significantly to this report through comments on the Securosis blog and follow-on review and conversations:

Tadd Axon

Mark Bower — Voltage

Drew Dillon

Jay Jacobs

Ulf Mattsson — Protegrity

Martin McKeay

Robert McMillon — RSA

Gary Palgon — NuBridges

thatdwayne

Branden Williams

Lucas Zaichkowsky

Sue Zloth

Copyright

Table of Contents

Introduction

One of the most daunting tasks in information security is protecting sensitive data in enterprise applications, which are often complex and distributed. Even the most hardened security professionals enters these projects with at least a modicum of trepidation. Coordinating effective information protection across application, database, storage, and server teams is challenging under the best of circumstances — and much tougher when also facing the common blend of legacy systems and conflicting business requirements.

We have generally addressed this issue with various forms of encryption, but over the past few years we have seen increasing interest in, and adoption of, *tokenization*.

Encryption, implemented properly, is one of the most effective security controls available to us. It renders information unreadable except to authorized users, and protects data both in motion and at rest. But encryption isn't the only data protection option, and there are many cases where alternatives make more sense. Sometimes the right choice is to remove the data entirely.

Tokenization is just such a technology: it replaces the original *sensitive* data with non-sensitive placeholders. Tokenization is closely related to encryption — they both mask sensitive information — but its approach to data protection is different. With encryption we protect the data by scrambling it using a process that's reversible if you have the right key. Anyone with access to the key and the encrypted data can recreate the original values.

With tokenization the process is not reversible. Instead we substitute a token value that's only associated with the 'real' data within a well-protected database. This surrogate can even have the exact same format (size and structure) as the original value, which helps minimize application changes. But the token itself is effectively meaningless, rather than a scrambled version of the original data, and thus cannot be compromised to reveal sensitive data.

The power of tokenization is that although *the token is usable within its native application environment, it is completely useless elsewhere*. So tokenization is ideal to protect sensitive identifying information such as credit card numbers, Social Security Numbers, and the other Personally Identifiable Information bad guys tend to steal and use or sell on the underground market. Unless they crack the tokenization server itself to obtain the original data, stolen tokens are worthless — and if they do, the tokens are redundant, because they'll have all the sensitive data anyway...

Interest in tokenization has picked up recently because it protects data at a potentially lower overall cost. Adding encryption to systems — especially legacy systems — introduces a burden outside the original design. Making application changes to accommodate encrypted data can dramatically increase overhead, reduce performance, and expand the demands on programmers and systems management staff. In distributed application environments the need to encrypt, decrypt, and re-encrypt data in different locations results in exposures that attackers can take advantage of. More instances where systems handle keys and data mean more opportunities for compromise. Key proliferation is

particularly dangerous due to *memory parsing malware*: malicious software capable of extracting keys or sensitive data directly from RAM, even without administrative privileges.

Aside from minimizing application changes, tokenization also reduces potential data exposure. Properly implemented, tokenization enables applications to use the token throughout the whole system, only accessing the protected value when absolutely necessary. Applications can use, store, and transact with the token without exposing the sensitive data it represents. Although some uses require the real (original) value, tokenization allows you to constrain its usage to your most secure implementations.

For example, one of the most common uses for tokenization is credit card transaction systems. We'll go into more depth later, but using a token for the credit card number allows us to track transactions and records without risk of stolen tokens being used for fraudulent charges. Access to the real number is limited to when we need to transmit a transaction to the payment processor. And if the *processor* uses tokenization as well, it's possible to completely eliminate stored credit card numbers.

This doesn't mean tokenization is always a better choice than encryption. They are closely related and the trick is to determine which will work best under the particular circumstances. For example, every tokenization system relies on encryption to protect sensitive data at rest. Just keep in mind that sometimes the best data security choice is to avoid keeping the data at all. Tokenization lets us remove sensitive data for some services and data stores while retaining most of its value.

In this report we dig deep into tokenization to explain how the technology works, explore different use cases and deployment scenarios, and review selection criteria to pick the right option. We'll cover everything from tokenization services for payment processing and PCI compliance to rolling your own solution for internal applications. Before we dig into the specifics, it's important to define a few terms that we use throughout this report.

Terms and Definitions

- **Token:** A random string used as a surrogate or proxy for some other data. There is no direct mathematical relationship between the original value and the random token, so the original data cannot be determined from the token. The association between the token and the original value is maintained in a database, and there is no other way to connect the two.
- **Tokenization:** The act of creating a token.
- **Encryption:** Transforming data into an unreadable state. For any given value the process consistently produces the same result, and the process can only be reversed with special knowledge (or the key).
- **Hash:** An irreversible mathematical function that scrambles data, producing a consistent and unique value. Used to create non-random tokens with some systems, but risks the original value being exposed through the mathematical relationship (if you can generating a matching hash).
- **Encrypted Token:** Data that has been encrypted, rather than replaced with a random value, using special encryption techniques that preserve format and data type. While these encrypted values are commonly used as tokens, each can be reversed into its original value, so this is technically encryption rather than tokenization.

Business Justification

Justifying an investment in tokenization entails two separate steps — first deciding to protect the data, and then choosing tokenization as the best fit for requirements. Covering all possible justifications for protecting data is beyond the scope of this paper, but the following drivers are typical:

- Meeting compliance requirements
- Reducing compliance costs
- Threat protection
- Risk mitigation

Once you have decided to protect the data, the next step is to pick the best method. Tokenization is designed to solve a very narrow but pervasive and critical problem: protecting discrete high-value data fields within applications, databases, storage, and across networks.

The most common use for tokenization is to protect sensitive key identifiers, such as credit card numbers, Social Security Numbers, and account numbers. Less frequently we see it used to protect full customer/employee/personnel records. The difference between the two (which we'll delve into during our architectural discussion) is that in the first case the tokenization server only stores the token and the original sensitive data, while in the second case it may include additional data such as names and addresses.

Reasons to select tokenization include:

- **Reduced compliance scope and costs:** Because tokenization completely replaces sensitive values with random values, systems that use the token instead of the real value are often exempt from audits and assessments required by regulations for systems which access the original sensitive data. For example, if you replace a credit card number in your application environment with tokens, systems using the tokens may be excluded from your PCI assessment — reduces assessment scope, and thus duration and cost as well.
- **Reduction of application changes:** Tokenization is often used to protect sensitive data within legacy application environments where we might otherwise use encryption. Tokenization allows us to protect the sensitive value with an analogue using the exact same format, which can minimize application changes. For example, encrypting a Social Security Number involves not only managing the encryption, but changing everything from form field logic to database field format requirements. Many of these changes can be avoided with tokenization, so long as the token formats and sizes match the original data. Tokens need not preserve formats and data types of the original values, but this capability is a key feature for reducing deployment costs.
- **Reduction of data exposure:** A key advantage of tokenization is that it *requires* data consolidation. Sensitive values are only stored on the tokenization server(s), where they are encrypted and highly protected. This reduces exposure over traditional encryption deployments, where cryptographic access to sensitive data tends to show up in many locations.

- **Masking by default:** The token value is random, so it also effectively functions as a data mask. You don't need to worry about adding masking to applications, because the real value is never exposed except where even the token could be misused within your environment. Tokenization solutions do not offer as many formatting options to preserve values for reporting and analytics as classic masking, but fully tokenized solutions provide greater security and less opportunity for data leakage or reverse engineering.

The most common reason organizations select tokenization over alternatives is cost reduction: reduced costs for application changes, followed by reduced audit/assessment scope. We also see organizations select tokenization when they need to update security for large enterprise applications — as long as you have to make a lot of changes, you might as well reduce potential data exposure and minimize the need for encryption at the same time.

Tokenization Process Overview

Tokenization is conceptually fairly simple. You merely substitute a marker of limited value for something of greater value. The token isn't *completely* valueless — it is important within its application environment — but its value is limited to the environment, or even a subset of that environment.

Think of a subway token or a gift card. You use cash to purchase the token or card, which then has value in the subway system or retail outlet. That token typically has a one-to-one relationship with the currency used to purchase it, but it's only usable on *that* subway system or in *that* retail chain. It still has value, we've just restricted *where* it has value.

Tokenization in applications and databases accomplishes the same thing. We take a generally useful piece of data, such as a credit card or Social Security Number and convert it to a local token that's useless outside the application environment designed to accept it. Someone might be able to use the token within your environment if they completely exploit your application, but they can't then use it anywhere else. This reduces risk significantly, as well as the scope of compliance requirements for the sensitive data.

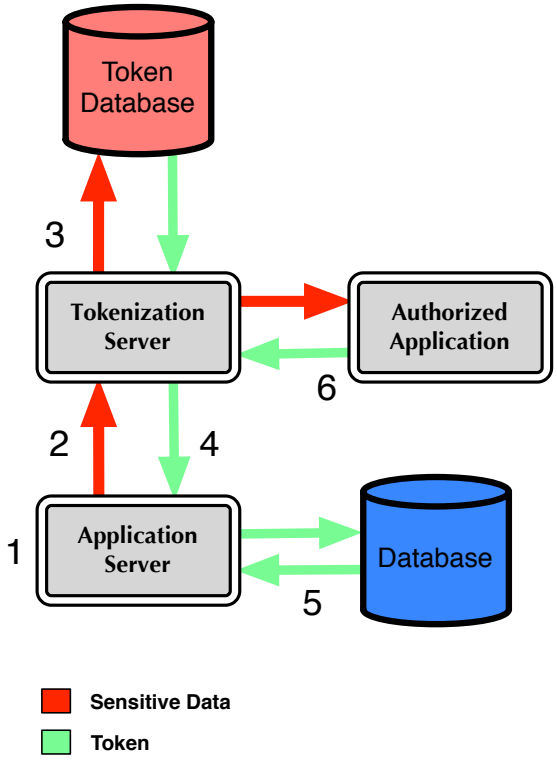
Basic Architectural Model

1. The application collects or generates a piece of sensitive data.
2. The data is immediately sent to the tokenization server — it is **not** stored locally.
3. The tokenization server generates the random or semi-random token. The sensitive value and the token are stored in a highly secure and restricted database (usually encrypted).
4. The tokenization server returns the token to the application.
5. The application stores the token, rather than the original value. The token is used for most transactions with the application.
6. When the sensitive value is needed, an authorized application or user can request it. The value is never stored in any local databases, and in most cases access is highly restricted. This dramatically limits potential exposure.

For this to work you must ensure a few things:

1. That there is no way to reproduce the original data without the tokenization server. This is different than encryption, where you can use a key and the encryption algorithm to recover the value anywhere.
2. All communications are encrypted.

Basic Tokenization Architecture



3. The application never stores the sensitive value — only the token.

4. Ideally the application never even touches the original value — as we will discuss later, there are architectures and deployment options to split responsibilities; for example, having a non-user-accessible transaction system with access to the sensitive data separate from the customer-facing side. You can have one system collect the data and send it to the tokenization server, another handle day-to-day customer interactions, and a third for transactions where the real value is needed.

5. The tokenization server and database are highly secure. Modern implementations are far more complex and effective than a locked-down database with both sets of values stored in a table.

But in the end it all comes down to the basics — take something widely valuable and replace it with a token of restricted value.

Tokens

Token generation and structure affect the security of the data, the ability to use the tokens as surrogates in other applications, and the overall performance of the system. In order to differentiate between the various solutions, it's important to understand the basics of how tokens are created.

Let's recap the process quickly. Each time sensitive data is sent to the token server, three basic steps are performed. First a token is created. Second the token and its corresponding original data are stored together in the token database. Third the token is returned to the calling application. The goal is to protect sensitive data *without sacrificing functionality* within applications, so we cannot simply create random blobs of data. The token format usually needs to match the format of the original data so it can be used as if it were the original (sensitive) data. For example, a Social Security token must have at least the same size (and likely data type) as a Social Security Number. Supporting applications and databases can accept the substituted value as long as it matches the constraints specified for the original value.

Token Creation

There are three common methods for creating tokens, only one of which we typically recommend:

- **Random Number Generation:** This method replaces data with a random number or alphanumeric value, and is our recommended method. Completely random tokens offers the greatest security, as the content cannot be reverse engineered. Some vendors use sequence generators to create tokens, grabbing the next value in the series to use for the token — this is not nearly as secure as a fully random number, but is very fast and secure enough for most (non-PCI) use cases. A major benefit of random numbers is that they are easy to adapt to any format constraints (discussed in greater detail below), and can be generated in advance to improve performance.

- **Encryption:** This method generates a 'token' by encrypting the data. Sensitive information is padded with a random salt to prevent reverse engineering, and then encrypted with the token server's private key. The advantage is that the 'token' is reasonably secure from reverse engineering, but the original value can be retrieved as needed. The downsides, however, are significant — performance is very poor, Format Preserving Encryption algorithms are required, and data can be exposed if keys are compromised or guessed. Further, the PCI Council, awaiting NIST certification, has not officially accepted format preserving cryptographic algorithms. Regardless, many large and geographically dispersed organizations that require access to original data favor the utility of encrypted 'tokens', even though this isn't really tokenization.

- **One-Way Hash Function:** Hashing functions create tokens by running the original value through a irreversible mathematical operation. This offers reasonable performance, and tokens can be formatted to match any data type. Like encryption, hashes must be created with a cryptographic salt (random bits of data) to thwart dictionary attacks. Unlike encryption, tokens created through hashing are not reversible. Security is not as strong as fully random tokens; but security, performance, and formatting flexibility are all improved over encryption.

Be aware that some open source and commercial token servers use poor token generation methods of dubious value. Some use reversible masking and others use unsalted encryption algorithms, and thus can be easily compromised and defeated.

Token Properties

Token solutions need to be flexible enough to handle multiple formats for the sensitive data they accept — such as personally identifiable information, Social Security Numbers, and credit card numbers. In some cases additional format constraints must be honored. As an example, a token representing a Social Security Number in a customer service application may need to retain the *real* last 4 digits. This enables customer service representatives to verify user identities without access to the rest of the SSN.

When tokenizing credit cards, tokens are the same size as the original credit card number — most implementations even ensure that tokens pass the LUHN check. As the token still resembles a real card number, systems that use the numbers need not to be altered to accommodate tokens. But unlike the real credit card or Social Security Numbers, tokens cannot be used as financial instruments, and have no value other than as a reference to the original transaction or real account. The relationship between a token and a card number is unique for any given payment system, so even if someone compromises the entire token database sufficiently that they can commit transactions in that system (a rare but real possibility), the numbers are worthless outside the environment they were created for. And most important, real tokens cannot be decrypted or otherwise restored back into the original credit card number.

Each data type has different use cases and tokenization vendors offer various generation options to accommodate them.

Token Datastore

Tokens, along with the data they represent, are stored within heavily secured databases with extremely limited access. The data is typically encrypted to ensure sensitive data is not lost in the event of a database compromise or stolen media. The token (database) server is the only point of contact with any transaction system, payment system, or collection point to reduce risk and compliance scope. Access to the database is highly restricted, with administrative personnel denied read access to the data, and even authorized access to the original data limited to carefully controlled circumstances.

Tokens are used to represent the same data for multiple events, possibly across multiple systems, so most token servers can issue *different* tokens for the same user data. A credit card number, for example, may get a different unique token for each transaction. The token server not only generates a new token to represent the new transaction, but is responsible for storing many tokens per user ID. Many use cases require that the token database support multiple tokens for each piece of original data, a one-to-many relationship. This provides better privacy and isolation, if the application does not need to correlate transactions by card number. Applications that rely on the sensitive data (such as credit card numbers) to correlate accounts or other transactions will require modification to use data which remains available (such as a non-sensitive customer number).

Token servers may be internally owned and operated, or provided as a third party service. We will discuss deployment models later.

Token Storage in Applications

When the token server returns the token to the application, the application must store the token safely, and effectively erase the original sensitive data. This is critical — not just to secure sensitive data, but also to maintain transactional consistency. Ideally the application should never store the original value — simply pass it through memory on its way to

the tokenization server. An interesting side effect of preventing reverse engineering is that a token by itself is meaningless — it only has value in relation to some other information. The token server has the ability to map the token back to the original sensitive information, but is the *only* place this can be done. Supporting applications need to associate the token with something like a user name, transaction ID, merchant customer ID, or some other identifier. This means applications that use token services must be resilient to communications failures, and the token server must offer synchronization services for data recovery.

This is one of the largest potential weaknesses: whenever the original data is collected or requested from the token database, it might be exposed in memory, log files, or virtual memory. This is the most sensitive part of the architecture, and later we'll discuss some of the many ways to split functions in order to minimize risk.

At this point you should have a good grasp of how tokens are generated, structured, and stored. In our next sections we will dig deeper into the architecture as we discuss tokenization server requirements and functions, application integration, and some sample architectural use cases.

Token Servers

Tokenization servers do a lot more than simply create and store tokens; among other functions they encrypt data, validate users, and return sensitive data as necessary and authorized. Many of these services are completely invisible to end users of token systems, and for day-to-day use you don't need to worry about the details. But the token server's internal design has significant effects on performance, scalability, and security. You need to assess these functions during selection to ensure you don't run into problems down the road.

For simplicity we will use credit card numbers as our primary example in this section, but as we've discussed any type of data can be tokenized. To better understand the functions performed by the token server, let's recap the two basic service requests. First, the token server accepts sensitive data (*e.g.,* credit card numbers) from authenticated applications and users, responds by returning a new or existing token, and stores the token with the encrypted value when creating new tokens. This comprises the bulk of all token server requests. Second, the token server returns decrypted information to approved applications when presented a token with acceptable authorization.

Authentication

Authentication is core to the security of token servers, which need to authenticate connected applications as well as specific users. To rebuff potential attacks, token servers should perform bidirectional authentication of all applications prior to servicing requests. The first step in this process is to set up a mutually authenticated SSL/TLS session, and verify that the connection was started with a trusted certificate from an approved application. Any strong authentication should be sufficient, and some implementations may layer additional requirements on top.

The second phase of authentication is to validate the user who issues a request. This may be a system/application administrator using specific administrative privileges, or one of many service accounts assigned privileges to request tokens or to request a given unencrypted credit card number. The token server provides separation of duties through these user roles: serving requests only from only approved users, through allowed applications, from authorized locations. The token server may further restrict transactions, perhaps only allowing a limited subset of database queries.

Data Encryption

Although technically the sensitive data might not be encrypted by the token server in the token database, in practice every implementation we are aware of encrypts the content. That means that prior to being written to disk and stored in the database, the data must be encrypted with an industry-accepted strong encryption cipher. After the token is generated, the token server encrypts the credit card number with a specific encryption key used only by that server. The data is then stored in the database, and thus written to disk along with the token for safekeeping.

Every current tokenization server is built on a relational database. These servers logically group tokens, credit cards, and related information in database rows — storing related items together. At this point, one of two encryption options is applied: either field level or transparent data encryption. In field level encryption just the row (or specific fields within it) is

encrypted. This allows a token server to store data from different applications (*e.g.,* credit cards from a specific merchant) in the same database with different encryption keys. Some token systems leverage transparent database encryption (TDE), which encrypts the entire database under a single key. In these cases the database performs the encryption on all data prior to writing to disk. Both forms of encryption protect data from indirect exposure such as someone examining disks or backup media, but field level encryption enables greater granularity (with a potential performance cost) and is more commonly used.

Finally, some implementations use asymmetric encryption to protect the data as it is collected within the application or point of sale device and sent to the server. The data is encrypted with the server's public key. The connection session will still typically be encrypted with SSL/TLS as well, but to support authentication rather than for additional security. The token server becomes the back-end point of decryption, using the private key to recover the plaintext prior to generating the token.

Key Management

Whether you use encrypted tokens or secure data at rest within the database, the token server uses encryption, and any time you have encryption you need key management. Key services may be provided directly from the vendor of the token services in a separate application, or by hardware security modules (HSM) if supported. Either way, keys are kept separate from the encrypted data and algorithms, providing security in case the token server is compromised, as well as helping enforce separation of duties between various system administrators. Each token server will have one or more unique keys to encrypt credit card numbers and other sensitive data. Symmetric keys are used, meaning the same key is used for both encryption and decryption. Communication between the token and key servers is mutually authenticated and encrypted.

Tokenization systems also need to manage any asymmetric keys for connected applications and devices. As with any encryption, the key management server/device/functions must support secure key storage, rotation, backup/restore, and logging.

Token Storage

Token storage is one of the more complicated aspects of token servers. How tokens are used to reference sensitive data or previous transactions is a major performance concern. Some applications require additional security precautions around the generation and storage of tokens, so tokens are not stored in a directly referenceable format. Use cases such as financial transactions with either single-use or multi-use tokens can require convoluted storage strategies to balance security of the data against referential performance. Let's dig into some of these issues:

- **Multi-use tokens:** Some systems provide a single token to reference every instance of a particular piece of sensitive data. So a credit card used at a specific merchant site will be represented by a single token regardless of the number of transactions performed. This is called a "multi-use token". This one to one mapping of data to token is easy from a storage standpoint; only one token for potentially hundreds of transactions. But while this model reduces storage requirements, it's not so great for security or fine grained differentiation between one instance of sensitive data and another. There are many use cases for creating more than one token to represent a single piece of sensitive data, such as anonymizing patient data across different medical record systems and credit cards used in multiple transactions with different merchants. Most token servers support the multiple-token model, if needed, enabling an arbitrary number of tokens to map to a given piece of data entity.

- **Token lookup:** Using a token to locate the credit card data is fairly straightforward; because the token is used to reference the (normally encrypted) sensitive data. But for multi-use tokens, how do you determine if a token has already been issued for a given credit card? Most ciphers use special modes of operation to avoid pattern leakage (*e.g.,* clues about the data that is encrypted), so each time you encrypt a credit card you get a different result. That means you cannot use the encrypted incoming Primary Account Number (PAN) as an index to determine whether a

token has already been issued for that card. There are three possible ways around this issue. One option is to create a hash of the PAN and store it with the encrypted credit card *and* the token; the hash serves as the index for the lookup. The second option is to turn off the special block encryption features (ECB, CBC, or Initialization Vector Randomization) that randomize the encryption output. Now you can compare the new encrypted value against the table to see if there is a match, but there is no free lunch: the encrypted output is *much* less secure as it's subject to simple dictionary attacks. Finally you can choose single-use tokens, issuing a new token for each transaction. Since not all schemes work well for each use case, you will need to investigate whether the vendor's performance and security are sufficient for your application.

- **Token collisions:** Token servers deployed for credit card processing have several constraints: they must keep the same basic format as the original credit card, and most expose the real last four digits. Some merchants demand that tokens pass LUHN checks. This creates an issue, as the number of tokens that meet these criteria is limited. The number of LUHN-valid 12-digit numbers creates a likelihood of the same token being issued — especially in single-use token implementations. Investigate your vendor's precautions to avoid or mitigate token collisions.

Architecture & Integration

When selecting a tokenization solution it is important to understand the primary architectural models, how token servers are deployed, how other applications communicate with token servers, and integration with supporting systems.

Architecture

There are three basic ways to build a token server:

1. Stand-alone token server with a supporting back-end database.

2. Embedded/integrated within another software application.

3. Fully implemented within a database.

Most commercial tokenization solutions are stand-alone software applications that connect to a dedicated database for storage, with at least one vendor bundling their offering into an appliance. All the cryptographic processes are handled within the application (outside the database), and the database provides storage and supporting security functions. Token servers use standard Database Management Systems, such as Oracle and SQL Server, but locked down very tightly for security. These may be on the same physical (or virtual) system, on separate systems, or integrated into a load-balanced cluster. In this model (stand-alone server with DB back-end) the token server manages all the database tasks and communication with outside applications. Direct connections to the underlying database are restricted, and cryptographic operations occur within the tokenization server rather than the database.

In an embedded configuration the tokenization software is embedded into the application and supporting database. Rather than introducing a token proxy into the workflow of credit card processing, existing application functions are modified to implement tokens. To users of the system there is very little difference in behavior between embedded token services and a stand-alone token server, but on the back end there are two significant differences. First, this deployment model usually involves some code changes to the host application to support storage and use of tokens. Second, each token is only useful for one instance of the application. Token server code, key management, and storage of the sensitive data and tokens all occur within the application. The tightly coupled nature of this model makes it very efficient for small organizations but does not support sharing tokens across multiple systems, and large distributed organizations may find performance inadequate.

Finally, it's possible to manage tokenization completely within a database without external software. This option relies on stored procedures, native encryption, and carefully designed database access & authorization controls. Used this way, tokenization is very similar to most data masking technologies. The database automatically parses incoming queries to identify and encrypt sensitive data. The stored procedure creates a random token — usually from a sequence generator within the database — and returns the token as the result of the user query. Finally all associated data is stored in a database row. Separate stored procedures are used to access encrypted data. This model was common before the

advent of commercial third-party tokenization tools, but has fallen into disuse due to its lack of advanced security features and failure to leverage external cryptographic libraries & key management services.

There are a few more architectural considerations:

- External key management and cryptographic operations are typically an option with any of these architectural models. This allows you to use more secure hardware security modules if desired.
- Large deployments may require synchronization of multiple token servers in different physically distributed data centers. This support must be built into the token servers, and is not available in all products. We will discuss this further when we get to usage and deployment models.
- Even when using a stand-alone token server, you may also deploy software plug-ins to integrate and manage additional databases that connect to the token server. This doesn't convert the database into a token server as we described in our second option above, but supports secure communication with distributed systems that need access to either the tokens or the protected data.

Integration

Tokenization must be integrated with a variety of databases and applications, so there are three ways to communicate with the token server:

1. **Application API calls:** Applications make direct calls to the tokenization server procedural interface. While at least one tokenization server requires applications to explicitly access its tokenization functions, this is now rare. Because of the complexity of the cryptographic processes and the need for proper use of the tokenization server; vendors now supply software agents, modules, or libraries to support integration of token services. These reside on the same platform as the calling application; rather than recoding applications to use the API directly, these modules support existing communication methods and data formats. This reduces code changes to existing applications, and provides better security — especially for application developers who are not security experts. These modules format the data for the tokenization APIs and establish secure communications with the tokenization server. This is generally the most secure option, as the code includes any required local cryptographic functions, such as encrypting new data with the token server's public key.
2. **Proxy Agents:** Software agents that intercept database calls (for example, by replacing an ODBC or JDBC component). In this model the process or application that sends sensitive information may be entirely unaware of the token process. It sends data as it normally does, and the proxy agent intercepts the request. The agent replaces sensitive data with a token and then forwards the altered data stream. These reside on the token server or its supporting application server. This model minimizes application changes, as you only need to replace the application/database connection and the new software automatically manages tokenization. But this does create potential bottlenecks and failover issues, as it runs in-line with existing transaction processing systems.
3. **Standard database queries:** The tokenization server intercepts and interprets the database requests. The SQL is parsed and sensitive data is substituted, then the query is run normally. This is potentially the least secure option, especially for ingesting content to be tokenized, but very simple to implement.

While this all sounds complex, there are really only two main functions to implement:

1. Send new data to be tokenized and retrieve the token.
2. When authorized, exchange the token for the protected data.

The server itself should handle pretty much everything else.

Deployment & Management

We have covered the internals of token servers and talked about architecture & integration of token services. Now we need to look at some of the different deployment models and how they match up to different types of businesses. Protecting medical records in multi-company environments is a very different challenge than processing credit cards for thousands of merchants.

Central Token Server

The most common deployment model we see today is a single token server that sits between application servers and the back-end transaction servers. The token server issues one or more tokens for each instance of sensitive information that it receives. For most applications it becomes a reference library, storing sensitive information within a repository and providing an index back to the real data as needed. The token service is placed in line with existing transaction systems, adding a new substitution step between business applications and back-end data processing.

As mentioned previously, this model is excellent for security as it consolidates all the sensitive data into a single highly secure server; additionally, it is very simple to deploy as all sensitive services reside in a single location. Limiting the number of locations which store and access sensitive data both improves security and reduces audit scope, as there are fewer systems with sensitive data to review.

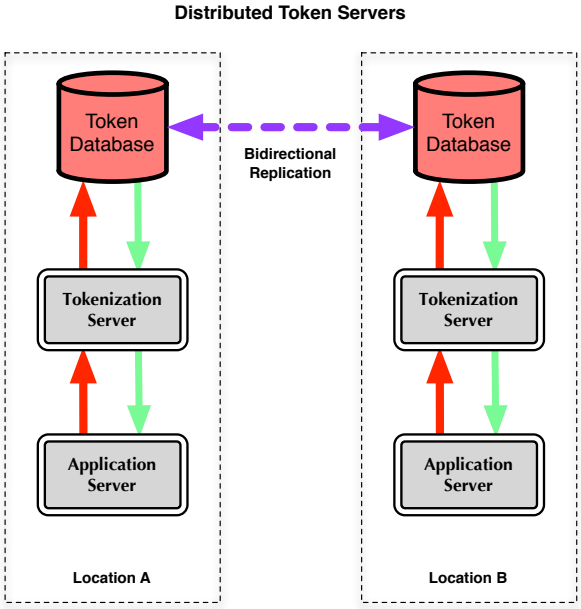
A central token server works well for small businesses with consolidated operations, but does not scale well for larger distributed organizations. Nor does it provide the reliability and uptime demanded by always-on Internet businesses. For example:

- **Latency:** The creation of a new token, lookup of existing customers, and data integrity checks are computationally complex. Vendors have worked hard to alleviate this problem, but some still have latency issues that make them inappropriate for certain operations or transaction support.
- **Failover:** If the central token server breaks down, or is unavailable because of a network outage, all processing of sensitive data (such as orders) stops. Back-end processes that convert to or from tokens halt.
- **Geography:** Remote offices, especially those in remote geographic locations, suffer from network latency, routing issues, and Internet outages. Remote token lookups are slow, and both business applications and back-end processes suffer disproportionately in the event of disaster or prolonged network outages.

To overcome issues in performance, failover, and network communications, several other deployment variations are available from tokenization vendors.

Distributed Token Servers

With distributed servers, the token databases are copies and shared among multiple sites. Each has a copy of the tokens and encrypted data. In this model each site is a peer of the others, with full functionality.



This model solves some of the performance issues with network latency for token lookup, as well as failover concerns. Since each token server is a mirror, if any single token server goes down the others can share its load. Token generation overhead is mitigated, as multiple servers assist in generation, and distribution of requests balances the load. Distributed servers are costly but appropriate for financial transaction processing.

While this model offers the best option for uptime and performance, synchronization between servers requires careful consideration. Multiple copies means synchronization must be carefully orchestrated, particularly for multi-use tokens and fraud/misuse detection. Carefully timed updates of data between locations along with key management so encrypted credit card numbers can be accessed are part of the deployment planning process. Finally, with multiple databases all serving tokens, you increase the number of repositories that must be secured, maintained, and audited — and thus inevitably cost.

Replicated Token Servers

In a replicated deployment, a single token server is designated as 'active', and one or more additional token servers are 'passive' backups. In this model if the active server crashes or is unavailable a passive server becomes active until the primary token server comes back online and a connection can be re-established. This 'mirroring' model improves on the central single server model by providing failover support while minimizing data consistency problems. These replicas are normally at the same location as the primary, but may be distributed across other locations. This differs from the distributed model in that only one server is active at a time, and while they store the same information, they are not all peers of one another.

The replication model is designed to provide consistency of operation, which is critical for banking and medical token deployments: if a token server fails, the passive server takes over and becomes the active server and operations

continue as normal. Synchronization is easier in this model, as the passive server need only mirror the active server, and bi-directional or n-way synchronization is not required. Token providers leverage the back-end mirroring capabilities built into relational database engines to provide this capability.

Systems Management

Finally, as with any major application, the token server includes standard management functions, but they tend to have additional security requirements:

- User management, including authentication, access, and authorization — for user, application, and database connections. Additionally, most tokenization solutions include extensive separation of duties controls to limit administrative access to the protected data.
- Backup and recovery of the stored data, system configuration, and encryption keys if encryption is managed on the token server. The protected data is always kept encrypted for backups.
- Monitoring and logging — Audit logs, especially of system changes, administrative access, and encryption key operations (such as key rotation), must be part of any solution you choose, and need to be evaluated prior to purchase. Reports are required in order to meet regulatory mandates.

The Visa Tokenization Best Practices guide for tokenization also recommends monitoring to "detect malfunctions or anomalies and suspicious activities in token-to-PAN mapping requests." While this is not a requirement for regulatory compliance, it is a very good security practice. This applies to both token generation requests and requests for unencrypted data. Monitoring should record and analyze activity in relation to security and operations policies. Some systems block unauthenticated requests, or even requests that don't match acceptable network attributes. And systems should throttle requests if they detect probable dictionary or "rainbow table" attacks.

Comments on Visa's recommendations

In 2010 Visa issued recommendations for tokenization in payment systems. Due to the reach of this guidance we feel it prudent to directly address some of the issues raised.

The odds are that if you are saddled with PCI-DSS responsibilities, you will not write your own 'home-grown' token servers. Even if you select commercial software, understand that these recommendations are open enough that vendors can easily provide botched implementations and still meet PCI-DSS or Visa's guidelines. If you are only interested in getting systems out of scope, then any solution is fine because QSAs will accept them as meeting the guidelines. But if you are going to the trouble of implementing a token server, it's no more work to select one that offers strong security.

The security concerns are twofold: Visa does not advocate random numbers for tokens, nor do they provide guidance on data storage.

Need for Random Numbers

The principle behind tokenization is to substitute a token for a real (sensitive) value, so you cannot reverse engineer the token into PAN data. But when choosing a token creation strategy, you must decide whether you want to be able to retrieve the value outside the token database or not. If you will want to convert the token back to the original value outside the system, use encryption. But if you don't need to do this, there are better ways to secure PAN data.

Visa's first suggestion should have been simply to *use a random number*. If the output is not generated by a mathematical function applied to the input, it cannot be reversed to regenerate the original PAN data. The only way to discover PAN data from a random token is a (reverse) lookup in the token server database. Random tokens are simple to

generate, and the size and data type constraints are trivial. This should be the *default*, as most firms should neither need nor want PAN data retrievable from the token.

As for encryption, rather than suggest a "strong encryption cipher", why not take this a step further and recommend a one time pad? This is a perfect application for that kind of substitution, and one-time pads are as secure as anything else. We believe Visa did not suggest this because a handful of very large payment processors, with distributed operations, actually want to retrieve the PAN data in multiple locations. That means they need encrypted tokens, and to distribute the keys.

An Encrypted Value is Not a Token

If you encrypt the value, it's encryption, not tokenization! Encryption obfuscates, but a token **removes**, the original data.

Conceptually the major advantages of tokenization are:

1. The token cannot be reversed back to the original value.

2. The token maintains the same structure and data type as the original value.

While format preserving encryption can retain the structure and data type, it's still reversible back to the original if you have the key and algorithm. Yes, you can add per-organization salt, but this is still reversible encryption.

When evaluating solutions, be careful to understand your potential provider's use of terminology. Tokenization and encryption are distinct processes and technologies, even though they are closely related. If the token is an encrypted value it's encryption, not tokenization.

Hashing

If tokens are based on a hashing algorithm, verify that the tokens are 'salted' with some form of rotating value. This avoids dictionary and "rainbow table" attacks to reverse the token. Some vendors do not salt the value, or salt with a static number (*i.e.,* merchant ID), neither of which is particularly secure. Also ensure that the hashing algorithm used is approved by NIST or other standards bodies, and determine whether the vendor's version has been scrutinized by an expert third party. Very small mistakes in hashing implementation render the hashes insecure.

Token Distinguishability

How do you know that a token is a token? When a token mimics the form and data type of the original data it is meant to detect, can you differentiate between the two? Some tokens generated for PCI compliance leave the last four bits un-tokenized, and they pass a LUHN check. In these cases it is almost impossible to tell. If you need "Token Distinguishability" check with your vendor. Visa recommends this as an option for all token servers, but depending upon the types of tokens you implement, it may not be possible.

Encrypting the Data Store

For mid-to-large vendors, credit cards and token servers will invariably use a relational database. To secure sensitive data within the relational repository, you will need to encrypt it — the question is how. There are many different ways to deploy encryption to secure data at rest, but we consider only three appropriate for the PCI-DSS standard: application layer encryption, field level encryption, and database transparent encryption. As the name implies, application layer encryption encrypts data within the application prior to insertion into the database. Field level encryption encrypts a field value within the database. Transparent database encryption encrypts all data blocks prior to passing them to the operating system to be written to disk. Both should use key management services to ensure proper separation of duties. If you are only using a single database that is highly secured, transparent database encryption is suitable and incredibly simple to deploy. If

you have many applications that will access the token store, or rely upon replicated/redundant databases, then application layer encryption provides better security.

Use Cases

We have covered most of the relevant bits of technology for token server construction and deployment. Armed with that knowledge we can tackle the most important part of the tokenization discussion: use cases. Which model is right for your particular environment? What factors should be weighed in the decision? The following three uses cases cover most of the customer situations we are asked for advice on. As PCI compliance is currently the overwhelming driver for tokenization, our first two use cases focus on different options for PCI-driven deployments.

Mid-Sized Retail Merchant

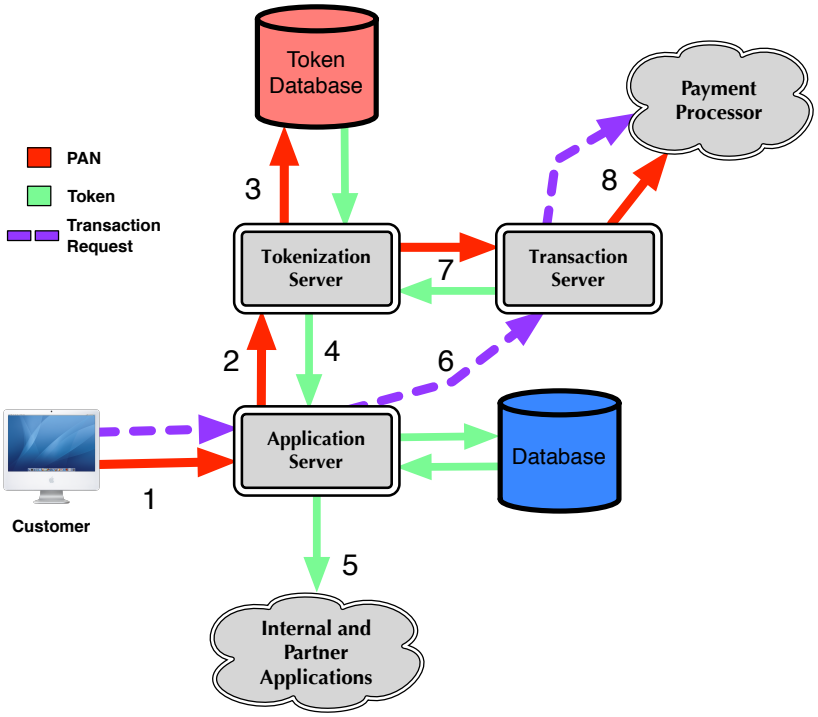
Our first case profiles a mid-sized retailer that needs to address PCI compliance requirements. The firm accepts credit cards but sells exclusively on the web, so they do not have to support point of sale terminals. Their focus is meeting PCI compliance requirements, and the question is how to achieve the goal at reasonable cost. As in many cases, most of the back office systems were designed before credit card storage was regulated, and use the CC# for customer and order identification. That means that order entry, billing, accounts receivable, customer care, and BI systems all store this number, in addition to web site credit authorization and payment settlement systems.

Credit card information is scattered across many systems, so access control and tight authentication are not enough to address the problem. There are simply too many access points to restrict with any certainty of success, and there are far too many ways for attackers to compromise one or more systems. Further, some back office systems are accessible by partners for sales promotion and order fulfillment. The security efforts will need to embrace almost every back office system and affect almost every employee. Most of the back office transaction systems have no particular need for credit card numbers — they were simply designed to store and pass the number as a reference value. The handful of systems that employ encryption are transparent, meaning they automatically return decrypted information and only protect data stored on disk or tape. Access controls and media encryption are not sufficient to protect the data or achieve PCI compliance in this scenario.

The principal project goal is PCI compliance, but as usual strong secondary goals are minimization of total costs, smooth integration, and keeping day-to-day management requirements under control. Because the obligation is to protect card holder data and limit the availability of credit cards in clear text, the merchant does have a couple options: encryption and tokenization. They could implement encryption in each of the application platforms or use a central token server to substitute tokens for PAN data at the time of purchase.

For our theoretical merchant we recommend in-house tokenization. An in-house token server will work with existing applications and provide tokens in lieu of credit card numbers. This will remove PAN data from the servers entirely with minimal changes to those few platforms that actually use credit cards: accepting them from customers, authorizing charges, clearing, and settlement — everything else will be fine with a non-sensitive token that matches the format of a real credit card number. We recommend a standalone server over one embedded within the applications, as the merchant will need to share tokens across multiple applications. This makes it easier to segment users and services authorized to generate tokens from those which legitimately need real credit card numbers.

Internal Credit Card Tokenization Architecture



1. A customer makes a purchase request. If this is a new customer, they send their credit card information over an SSL connection (which should go without saying). For future purchases, only the transaction request need be submitted.

2. The application server processes the request. If the credit card is new, it uses the tokenization server's API to send the value and request a new token.

3. The tokenization server creates the token and stores it with the encrypted credit card number.

4. The tokenization server returns the token, which is stored in the application database with the rest of the customer information.

5. The token is then used throughout the merchant's environment, instead of the real credit card number.

6. To complete a payment transaction, the application server sends a request to the transaction server.

7. The transaction server sends the token to the tokenization server, which returns the credit card number.

8. The transaction information — including the real credit card number — is sent to the payment processor to complete the transaction.

While encryption could protect credit card data without tokenization, and be implemented to minimize changes to UI and database storage in supporting applications, it would require modification of every system that handles credit cards. And a pure encryption solution would require support of key management services to protect encryption keys. The deciding factor against encryption here is the cost of retrofitting systems with application layer encryption — especially because several rely on third-party code. The required application revisions, changes to operations management and disaster recovery, and broader key management services required would be far more costly and time-consuming. Recoding applications would become the single largest expenditure, outweighing the investment in encryption or token services.

The goal is compliance and data security, but ultimately any merchant's buying decision is heavily affected by cost: for acquisition, maintenance, and management. And for any merchant handling credit cards, as the business grows so does the cost of compliance. Likely the 'best' choice will be the one that costs the least money today and in the long term. In terms of relative security, encryption and tokenization are roughly equivalent. There is no significant purchase cost difference between the two, either for acquisition or operation. But there is a significant difference in the expense of implementation and auditing for compliance.

Tokenization as a Service

In response to a mix of breaches and PCI requirements, some payment processors now offer tokenization as a service. Merchants can subscribe in order to avoid any need to store credit cards in their environment — instead the payment processor provides them with tokens as part of the transaction process. It's an interesting approach, which can almost completely remove the PAN (Primary Account Number) from the environment.

The trade-off is that this closely ties you to your processor, and requires you to use only their approved (and usually bundled) hardware and software. You reduce risk by removing credit card data entirely from your organization, at a cost in flexibility and (probably) higher switching costs.

Many major processors have built end-to-end solutions using tokenization, encryption, or a combination the two. For our example we will focus on tokenization within a fairly standard Point of Sale (PoS) terminal architecture, such as we see in many retail environments.

First a little bit on the merchant architecture, which includes three components:

1. Point of Sale terminals for swiping credit cards.

2. A processing application for managing transactions.

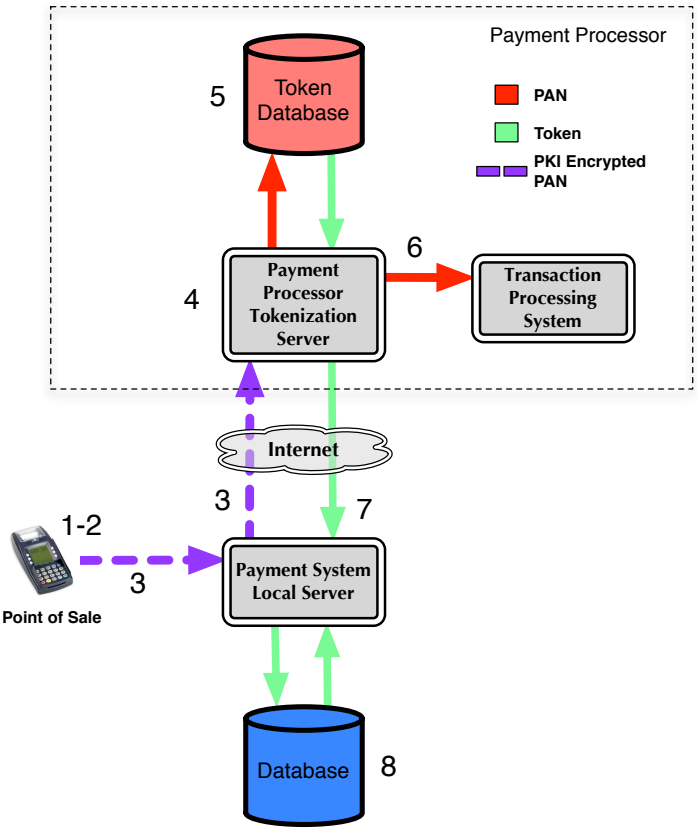
3. A database for storing transaction information.

Traditionally, a customer swipes a credit card at the PoS terminal, which then communicates with an on-premise server, which in turn connects either to a central processing server in the merchant's environment for payment authorization or batch clearing, or directly to the payment processor. Transaction information, including the PAN, is stored on the on-premise and/or central server. PCI-compliant configurations encrypt the PAN data in the local and central databases, as well as all communications.

When tokenization is implemented by the payment processor, the process changes to:

1. Retail customer swipes the credit card at the PoS.
2. The PoS encrypts the PAN with the public key of the payment processor's tokenization server.
3. The transaction information (including the PAN, other magnetic stripe data, the transaction amount, and the merchant ID) are encrypted and transmitted to the payment processor.
4. The payment processor's tokenization server decrypts the PAN and generates a token. If this PAN is already in the token database, they can either reuse the existing token (multi-use), or generate a new token specific to this transaction (single-use). Multi-use tokens may be shared among different vendors.
5. The token, PAN data, and possibly merchant ID are stored in the tokenization database.
6. The PAN is used by the payment processor's transaction systems for authorization and charge submission to the issuing bank.
7. The token is returned to the merchant's local and/or central payment systems, as is the transaction approval/denial, which hands it off to the PoS terminal.

Service Provider Credit Card Tokenization Architecture



8. The merchant stores the token with the transaction information in their systems/databases. For the subscribing merchant, future requests for settlement and reconciliation to the payment processor reference the token.

The key here is that the PAN is encrypted at the point of collection, and in a properly-implemented system is never again in the merchant's environment. The merchant never again has the PAN — they simply use the token in any case where the PAN would have been used previously, such as for processing refunds.

This is a fairly new approach and different providers use different options, but the fundamental architecture is fairly consistent.

Protecting Personally Identifiably Employee Information

Not every use case for tokenization involves PCI-DSS. There are equally compelling implementation options, several for personally identifiable information, that illustrate different ways to deploy token services. Here we will describe how tokens are used to replace Social Security numbers in human resources applications. These services must protect the SSN during normal use by employees and third party service providers, while still offering authorized access for human resources personnel as well as payroll and benefits services.

In our example an employee uses an HR application to review benefits information and make adjustments to their own account. Employees using the system for the first time will establish system credentials and enter their personal information, potentially including Social Security Number. To understand how tokens work in this scenario let's map out the process:

1. The employee account creation process is started by entering the user's credentials and adding personal information including the Social Security Number. This is typically performed by HR staff, with review by the employee in question.

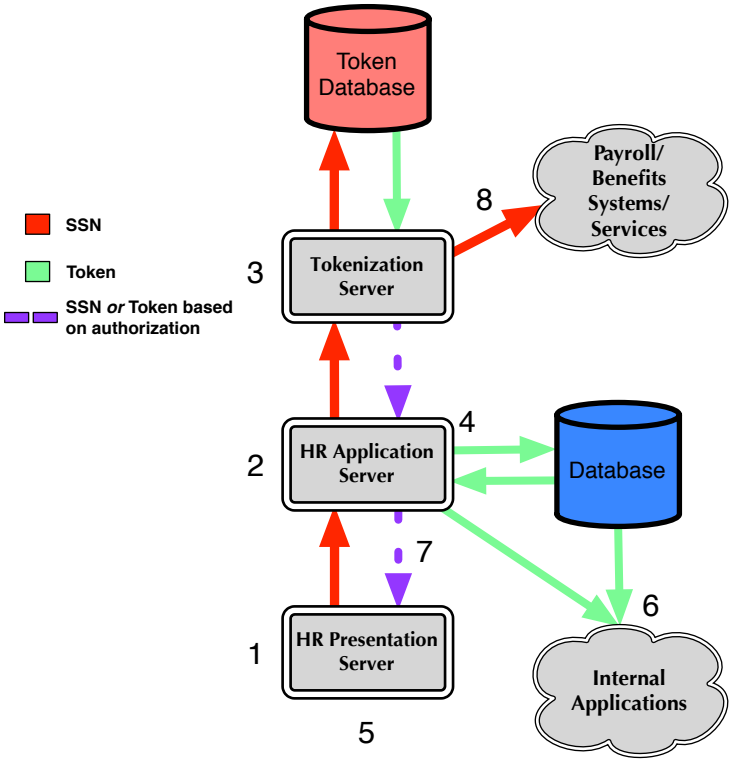
2. Over a secure connection, the presentation server passes employee data to the HR application. The HR application server examines the request, finds the Social Security Number is present, and forwards the SSN to the tokenization server.

3. The tokenization server validates the HR application connection and request. It creates the token, storing the token/Social Security Number pair in the token database. Then it returns the new token to the HR application server.

4. The HR application server stores the employee data along with the token, and passes the token back to the presentation server. The application server's temporary copy of the original SSN is overwritten so it does not persist in memory.

5. The presentation server displays the successful account creation page, including the tokenized value, back to the user. The original SSN is overwritten so it does not persist in token server memory.

HR/Benefits System Tokenization Architecture



6. The token is used for all other internal applications that might have previously relied on real SSNs.

7. Occasionally HR employees need to look up an employee by SSN, or access the SSN itself (typically for payroll and benefits). These personnel are authorized to see the real SSN within the application under the right conditions (this needs to be coded into the application using the tokenization server's API). Although the SSN shows up in their application screens when needed, it isn't stored on the application or presentation server. Typically it isn't difficult to keep the sensitive data out of logs, although it's possible SSNs will be cached in memory. That's a real risk, but far smaller than otherwise.

8. The real SSN is used as needed for connections to payroll and benefits services/systems. Usage is reduced as possible, but realistically many (most?) major software tools and services still require the SSN — especially for payroll and taxes.

Applications that already contain Social Security Numbers undergo a similar automated transformation process to replace the SSN with a token, and this occurs without user interaction. Many older applications used SSN as the primary key to reference employee records, so referential key dependencies make replacement more difficult and may involve downtime and structural changes.

Note than as surrogates for SSNs, tokens can be formatted to preserve the last 4 digits. Display of the original trailing 4 digits allows HR and customer service representatives to identify the employee, while preserving privacy by masking the first 5 digits. There is never any reason to show an employee their own SSN after initial configuration — they should already know it and can ask for assistance from HR if there's an actual need to verify the number on file — and non-HR personnel should never see SSNs either. The HR application server and presentation layers will only display the tokenized values to the internal web applications for general employee use, never the original data.

But what's really different about this use case is that HR applications need regular access to the original Social Security Number. Unlike a PCI tokenization deployment — where requests for original PAN data are somewhat rare — accounting, benefits, and other HR services regularly require the original sensitive data. Within our process, authorized HR personnel can use the same HR application server, through a HR specific presentation layer, and access the original Social Security Number. This is performed automatically by the HR application on behalf of validated and authorized HR staff, and limited to specific HR interfaces. After the HR application server has retrieved the employee information from the database, the application instructs the token server to get the Social Security Number, and then sends it back to the presentation server.

Similarly, automated batch jobs such as payroll deposits and 401k contributions are performed by HR applications, which in turn instruct the token server to send the SSN to the appropriate payroll/benefits subsystem. Social Security Numbers are retrieved by the token server and then passed to the supporting application over a secure authenticated connection. In this case, the token appears at the presentation layer, while authorized third party providers receive the SSN via proxy on the back end.

Selection Criteria

If you are looking at tokenization we can assume you want to reduce exposure of sensitive data while saving some money by curtailing security requirements across your IT operation. We don't want to understate the complexity of tokenization, but the selection process itself is fairly straightforward. Ultimately there are just a handful of questions you need to address: Does this meet my business requirements? Is it better to use an in-house application or choose a service provider? Which applications need token services, and how hard will they be to set up?

For some organizations the selection process is fairly easy. If you are a small firm dealing with PCI compliance, choose an outsourced token service through your payment processor. It's likely they already offer the service, and if not they will soon. And the outsourced token service should easily integrate with your other external services, especially since they come from the service provider — at least something compatible and approved for their infrastructure. Most small firms simply do not possess the resources and expertise in-house to set up, secure, and manage a token server. Even with the expertise available, choosing a vendor-supplied option is cheaper and removes most of the liability from your end.

Using a service from your payment processor is actually a great option for any company that already fully outsources payment systems to its processor, although this tends to be less common for larger organizations.

The rest of you have some work to do. Here is our recommended process:

1. **Determine Business Requirements:** The single biggest consideration is the business problem to resolve. The appropriateness of a solution is predicated on its ability to address your security or compliance requirements. Today this is generally PCI compliance, so fortunately most tokenization servers are designed with PCI in mind. For other data such as medical information, Social Security Numbers, and other forms of PII, there is more variation in vendor support.

2. **Map and Fingerprint Your Systems:** Identify the systems that store sensitive data — including platform, database, and application configurations — and assess which contain data that needs to be replaced with tokens.

3. **Determine Application/System Requirements:** Now that you know which platforms you need to support, it's time to determine your specific integration requirements. This is mostly about your database platform, what languages your applications are written in, how you authenticate users, and how distributed your application and data centers are.

4. **Define Token Requirements:** Examine how data is used by your applications and determine whether single-use or multi-use tokens are preferred or required. Can the tokens be formatted to meet the business use defined above? If cleartext access is required in a distributed environment, are encrypted format-preserving 'tokens' suitable?

5. **Evaluate Options:** At this point you should know your business requirements, understand your particular system and application integration requirements, and have a grasp of your token requirements. This is enough to start evaluating the different options on the market, including services vs. in-house deployment.

It's all fairly straightforward, and the important part is to determine your business requirements ahead of time, rather than allowing a vendor to steer you toward their particular technology. Since you will be making changes to applications and databases it only makes sense to have a good understanding of your integration requirements before letting the first salesperson in the door.

There are a number of additional secondary considerations for token server selection.

- **Authentication:** How will the token server integrate with your identity and access management systems? This is a consideration for external token services as well, but especially important for in-house token databases, as the real PAN data is present. You need to carefully control which users can make token requests and which can request cleartext credit card or other information. Make sure your access control systems will integrate with your selection.

- **Security of the Token Server:** What features and functions does the token server offer for encryption of its data store, monitoring transactions, securing communications, and request verification. On the other hand, what security functions does the vendor assume you will provide?

- **Scalability:** How can you grow the token service with demand?

- **Key Management:** Are the encryption and key management services embedded within the token server, or does it depend on external key management services? For tokens generated via encryption, examine how keys are used and managed.

- **Performance:** In payment processing speed has a direct impact on customer and merchant satisfaction. Does the token server offer sufficient performance for responding to new token requests? Does it handle expected and unlikely-but-possible peak loads?

- **Failover:** Payment processing applications are intolerant of token server outages. In-house token server failover capabilities require careful review, as do service provider SLAs — be sure to dig into anything you don't understand. If your organization cannot tolerate downtime, ensure that the service or system you choose accommodates your requirements.

Who We Are

About the Authors

Rich Mogull, Analyst and CEO

Rich has twenty years experience in information security, physical security, and risk management. He specializes in data security, application security, emerging security technologies, and security management. Prior to founding Securosis, Rich was a Research Vice President at Gartner on the security team where he also served as research co-chair for the Gartner Security Summit. Prior to his seven years at Gartner, Rich worked as an independent consultant, web application developer, software development manager at the University of Colorado, and systems and network administrator. Rich is the Security Editor of TidBITS, a monthly columnist for Dark Reading, and a frequent contributor to publications ranging from Information Security Magazine to Macworld. He is a frequent industry speaker at events including the RSA Security Conference and DefCon, and has spoken on every continent except Antarctica (where he's happy to speak for free — assuming travel is covered).

Adrian Lane, Analyst and CTO

Adrian Lane is a Senior Security Strategist with 22 years of industry experience, bringing over a decade of C-level executive expertise to the Securosis team. Mr. Lane specializes in database architecture and data security. With extensive experience as a member of the vendor community (including positions at Ingres and Oracle), in addition to time as an IT customer in the CIO role, Adrian brings a business-oriented perspective to security implementations. Prior to joining Securosis, Adrian was CTO at database security firm IPLocks, Vice President of Engineering at Touchpoint, and CTO of the security and digital rights management firm Transactor/Brodia. Adrian also blogs for Dark Reading and is a regular contributor to Information Security Magazine. Mr. Lane is a Computer Science graduate of the University of California at Berkeley with post-graduate work in operating systems at Stanford University.

About Securosis

Securosis, L.L.C. is an independent research and analysis firm dedicated to thought leadership, objectivity, and transparency. Our analysts have all held executive level positions and are dedicated to providing high-value, pragmatic advisory services.

We provide services in four main areas:

- Publishing and speaking: including independent objective white papers, webcasts, and in-person presentations.
- Strategic consulting for end users: including product selection assistance, technology and architecture strategy, education, security management evaluation, and risk assessment.
- Strategic consulting for vendors: including market and product analysis and strategy, technology guidance, product evaluation, and merger and acquisition assessment.
- Investor consulting: technical due diligence, including product and market evaluations, available in combination with deep product assessment with our research partners.

Our clients range from stealth startups to some of the best known technology vendors and end users. Clients include large financial institutions, institutional investors, mid-sized enterprises, and major security vendors.

Securosis has partnered with security testing labs to provide unique product evaluations that combine in-depth technical analysis with high-level product, architecture, and market analysis.