Encrypting WebSphere Application Server system passwords

Martin Lansche

November 23, 2016

IBM WebSphere Application Server stores system passwords in files that are encoded. To support clients who want to implement their own password storage mechanism, WebSphere Application Server provides a system programming interface (SPI). If you require a password encryption solution, this article provides an example of how you can achieve it by using the SPI.

Introduction

Almost daily we are bombarded with stories in the technical and mainstream press about attacks of computer systems where passwords are stolen and then become available to attackers. The press frequently points out that one reason that this password data was retrievable is because "it was not encrypted." In this situation, these passwords in question are user passwords that are used to log in and access the system. They are not the passwords that are associated with the system processes and binary files. This difference is extremely important.

With such press coverage in mind, perhaps the most common comment centers around customers who want to encrypt passwords in their IBM® WebSphere® Application Server systems. This comment is often repeated by WebSphere security consultants, our technical sales people, and the WebSphere security development architects. The message that we get is, "WebSphere is mostly meeting its claim of being secure by default, but our passwords are stored on the filesystem that are encoded, not encrypted." This is frequently followed by "and we failed our security audit because they are not encrypted."

Some clients insist that encryption of system passwords that are used by WebSphere Application Server is preferred over encoding. However, a system programming interface (SPI) has been available since WebSphere Application Server Version 6.0.2, which can be used to implement any password "hiding" solution that a client might want to implement. By using this SPI, IBM Hybrid Cloud Services (formerly known as IBM Software Services for WebSphere) has implemented such a solution for several clients. In this article, I define some basic security concepts and then describe the design considerations in this custom solution.

The basics: Encoding versus encryption

What is the difference between encoding and encrypting? We use the term encoding to apply a scheme where something is hidden by formula or algorithm. Consider an example of a secret
password that you want to "hide." For example, suppose that I have an admittedly bad password: SecretMonkey. A reversible algorithm might be used to encode this phrase to hide its true value.

Julius Caesar is attributed to have used an encoding scheme that is called a Rotation cipher or a ROT cipher, where the numerical position in the alphabet is shifted by some number. History now calls this method a Caesar cipher. The ROT3 cipher maps three character positions to the right. The mapping for ROT3 (for an English alphabet) is:

Plain: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Cipher: DEFGHIJKLMNOPQRSTUVWXYZABC

My ROT3 ciphered password above then becomes:

Plain: SecretMonkey
Cipher: VhfuhwPrqnhb

When presented with this encoded string, professional cryptanalysts might quickly guess the encoding algorithm and decode it with the reversing algorithm (ROT-3). The strength of encoding is in the secrecy of the algorithm. Remember that attackers are adept at reverse-engineering algorithms. When that algorithm is known, the data is unprotected.

Encryption, as opposed to encoding, bases its security not on the algorithm that is used, but on the strength of hard-to-guess keys. Still, algorithms are used, but these algorithms are (typically) widely known and understood. Encryption algorithms differ from encoding algorithms in that they take two inputs: the text and a secret key. In much the same way that a 4-digit debit card PIN uses a 1 in 10,000 possible ($10^4$) choice secret, encryption keys use a secret key that takes a lot of effort to brute-force guess.

Two types of encryption are possible, and both types are based on complex mathematics. The first type of encryption involves a single key, called shared key, secret key, or symmetric key encryption.

Consider two functions:

cipherText = Encrypt(plainText, secretKey);

And, consider the reverse:

plainText = Decrypt(cipherText, secretKey);

To protect my data, I chose a key from a large possible set of keys, for example, a 256-bit key. This key is a set of $2^{256}$ possible numbers, which is a huge number; in decimal notation that is $1.15x10^{77}$: a 115 followed by 75 more digits. As you can imagine, it might take a long time to call the Decrypt function, iterating through all possible keys.

The second type of encryption uses two different, or asymmetric, keys. This type of encryption is known as public key or asymmetric encryption. These two keys are related to each other in a complex mathematical relationship. Without going into the mathematics, the function call to encrypt some text is:
cipherText = Encrypt(plainText, KeyA);

The asymmetric reverse is:
plainText = Decrypt(cipherText, KeyB);

The strength of both types of encryption rest on the difficulty of guessing the key argument that is needed for that decrypt function. In this case, that's either the shared "secretKey" for data encrypted using symmetric encryption, or the "KeyB" for asymmetric encryption.

**Password hashing**

I want to provide a little more theory here and introduce the idea of a one-way hash function. Again, I will not go into the mathematics, because the details are not important to my argument. A one-way hash function (again) uses a well-known and understood algorithm:
digest = hash(inputString);

Here, the **inputString** is transformed into an output that is called a **digest**, and it is computationally infeasible to reverse the process (therefore, the term **one-way**). Every time that you provide the same inputString, you get the same digest. The digest is usually a constant length for a particular hash function. Hash functions have the additional property that, if you change any bit in the input, you get a completely different digest output.

A trivial hash function can produce an 8-bit digest. Every input string produces one of $2^8$ digests. If any of the input changes, a different digest is produced. If you are still paying attention, you realize that an infinite number of input strings produces the same digest. But, we know that, from a digest, it is impossible to guess the inputString. When the size of the digest becomes big enough (for example, $2^{256}$ bits), the chances of another random input that produces the same digest are infinitely small.

Attackers realized that, for a known algorithm, it is possible to pre-generate all possible input and calculate their own digests, which were known as "rainbow" tables. In practice, implementers change the hash function behavior by concatenating any input string with a secret or different prefix called a **salt**. So, the preferred usage is:
digest = hash(inputString||salt);

An attacker would have to build rainbow tables with all possible **salt** strings.

Why, you might be asking, am I meandering, and not getting to my point? Those press reports of stolen password databases are talking about systems where the **users'** passwords are checked against a database, and sadly, that data in the database was not properly protected. It is well accepted knowledge in the IT field that those passwords should not be stored in plain text, but that they should contain the digest of the password with salt. That is, a password repository should not contain my password string **SecretMonkey**. When validating my password, the system should concatenate the particular salt that it uses to my string, compute the hash of that concatenated string, and compare it against the saved password digest. If those two digests match, there is an extremely high probability the inputString – **SecretMonkey** – was my password.
A fictitious example of hashing my string, with different salts might be:

- SecretMonkey (without salt) > !GHD%@DB&!DJD
- SecretMonkey||(salt=1111) > UCHAKJ$HDJS
- SecretMonkey||(salt=2222) > njurnasiurrijfdd

Your security audit must ensure that these passwords are stored in hashed form. Notice I did not say encrypted, but hashed. I hope you understand the subtle distinction, which is that one is reversible and the other is not.

But I am not finished. Without question, I support anyone who says that **user passwords must be stored as salted, hashed digests**. But, user passwords and system passwords have an important difference.

**User passwords are not system passwords**

A user password is used to prove or confirm the identity of a user, typically an end user. A system password (such as my example of **SecretMonkey**) is sent to another downstream system, along with a user ID, to represent an account to the downstream system.

When I log in to a financial application, I can imagine that the following happens:

- I am authenticated against the LDAP. My user ID and password are used in an LDAP bind operation. That LDAP server hashes the password (along with its unique salt) and compares the resulting digest against my saved digest to confirm or reject my claim of identity.
- When authenticated, the web application connects to a back-end databases to find new notifications and network requests. This information can be an SQL command similar to

```
select notifications where userid=Lansche
```

To prevent any computer within the network from having access to this data, the financial application connects to that database with FinancialApp credentials. For the sake of argument, imagine that these credentials are a user ID and password (**SecretMonkey**).

This simple example shows usage of a system password. Application servers need system passwords to perform LDAP searches, to connect to databases, queue managers, web services, and other applications, and for application runAs identities. When the application server authenticates to these other systems with a user ID and password, the password needs to be sent in a "usable" format — the plain-text version of the password. As my colleague T. Rob Wyatt explains in his blog post "Encrypting passwords in config files – secure or not?":

_Ultimately, it comes down to this: any system that must boot to operational status unattended must have access to a usable authentication credential ..."Unattended" means that the system must go from powered down to fully operational without human intervention. Is it acceptable that, after rebooting a server, all the applications start a command prompt waiting for a human user to enter all the passwords? Usually not. Overwhelmingly not. There are some cases where this is true but I don't have the security clearances to work with those systems and if I did I couldn't write about it anyway. For everyone else, we all need systems that recover automatically and without_
human intervention. Bottom line, "unattended" means that the credentials must be stored where the application can access them at run time, usually in the file system.

That is, when the Financial application starts, you have two choices:

- Wait for an operator at a console to type the password for the FinancialApp user ID (and every other system account needed).
- Have the application obtain the password from somewhere on the file system.

I am confident that real applications follow the second model. Most non-military and national-security systems follow the second model.

As received by the downstream system, this user ID and password must be in plain text. The downstream system (hopefully) follows best practices, computes a salted hash of the received password (njurnasiurrjfd as used in the previous example) and compares this value against its password repository.

As a thought experiment, imagine if WebSphere were to send the password pre-hashed, with salt, (njurnasiurrjfd) to the downstream system, and the downstream system just compared that received password against its repository. In that case, the password effectively becomes njurnasiurrjfd.

Stealing the downstream system's password repository might provide the attacker with the effective passwords because they are not protected with a one-way hash anymore. This experiment describes a worse situation than we started with.

At this point, I hope you understand why the caller (WebSphere in this case) must have the real plain text versions of the system passwords, at least at the moment when the actual call to the downstream system is made.

Is password encoding sufficient?

Let us analyze the process that the system now goes through logically.

1. Assume that encoding passwords is not sufficient. Why? Because anyone with read access to the file system has access to the credentials:
   - Perhaps users have access to the file system.
   - Applications running in WebSphere Application Server can read the passwords.
2. You implement the password encoding SPI, and encrypt the password. To do this, you need a key. The password is now stored encrypted on the file system. Assume a symmetric encryption. (It is faster than asymmetric encryption.) You store the key in a keystore.
3. To convert from the encrypted text back to plain text, the server needs the decryption key. The server reads the key from the keystore. To prevent any application from trivially getting that key from the keystore, you password protect the keystore. WebSphere Application Server needs access to that password to read the keystore. Where is that password saved? Because prompting an operator is not acceptable, that password is stored in a file on the file system. A
file that contains a password to enable an application to read a keystore is known as a *stash* file.

But wait! We said in step 1, that our fundamental issue is that we are unable to control which processes have access to the file system. We *still* have that problem. Now, it is more inconvenient. However, all the necessary pieces to get this system password remain on the file system and are accessible to WebSphere Application Server or applications that are running inside the application server:

1. The password as cipher text within the WebSphere Application Server configuration files.
2. The keystore that contains the key that is needed to decrypt that cipher text.
3. The stash file that contains the password to read the key from the keystore.

We have moved the problem two steps to the right. The keystore and stash file can be located in another location in the file system or in a hardware keystore, from which it cannot be easily exfiltrated. This method is more secure than the passwords that are only encoded in a file and that file within the WebSphere configuration directories.

**Hints:** Use file system access control, and carefully review all application code.

However, the real security issue to be addressed is controlling who (people or applications) has access to the file system. The best way to address this real security issue is discussed in *step 17 of Advanced security hardening in WebSphere Application Server V7, V8 and V8.5: Overview and approach to security hardening*. For many enterprises, it is likely that password encoding with appropriate application security and system access controls are sufficient. However, it is reasonable to consider taking the additional step of encrypting system passwords.

**Where encryption of system passwords can be helpful**

Do not infer that password encryption is not a good idea. Instead, in addition to infrastructure hardening, you must also consider weaknesses in people, in other applications beyond WebSphere, and in Java® Enterprise Edition (Java EE) applications that are deployed and running in WebSphere.

An administrator who posts configuration XML files to Stack Overflow inadvertently exposes the encoded passwords. Similarly, sending these XML files to IBM for support purposes exposes these files. It might be too much to ask our customers to obfuscate sensitive data in the files that they send us. In both of these cases, encrypted passwords might avoid accidental publishing or sharing of system passwords.

Any other application on your system might contain security bugs that make it possible to read and exfiltrate files. If that application can be run as the WebSphere user, or as the root user, and the files can be exfiltrated from the system, password encryption of the WebSphere files make it slightly more difficult to decrypt the passwords.

Application weaknesses can be the result of careless design or coding, or they can be the result of a malicious programmer. Depending on just how sloppy the programmer was, encryption of passwords might make the attackers’ job a lot harder or maybe just a little bit harder. If you have
malicious coders, who are deliberately trying to obtain passwords, password encryption will not stop them. They just use the same code that the container uses to decrypt the password.

**Defining your own password handler**

The interface for defining your own password handler is the `com.ibm.wsspi.security.crypto.CustomPasswordEncryption` interface. You create a `com.ibm.wsspi.security.crypto.CustomPasswordEncryptionImpl` class and place the class in the `<WAS>/lib/ext` directory. WebSphere Application Server processes use this class without further administrative configuration. Only a single class implements the interface. This interface defines the following methods (as expected).

```java
EncryptedInfo encrypt(byte[] decrypted_bytes)
    throws PasswordEncryptException
byte[] decrypt(EncryptedInfo info)
    throws PasswordDecryptException
void initialize(java.util.HashMap initialization_data)
```

This last method is reserved for future use and is currently not called by the WebSphere Application Server run time.

WebSphere Application Server interacts with these passwords in two ways:

- It stores the files when the passwords change.
- It reads the files to obtain the password.

When WebSphere Application Server stores a password that has been updated, it checks to see whether a custom encryption provider is available. If a provider is available, it will be given the cleartext password. The provider should manipulate the password in some way and return to WebSphere Application Server binary information that is meaningful to the provider. WebSphere Application Server then stores this binary information in an appropriate XML file as usual. The stored data indicates that it was encoded by using a custom provider, which impacts future reads.

When WebSphere Application Server needs a password, it reads the configuration information (stored as XML files) and examines the information to determine whether a custom provider was used. If the password is encoded in the default manner, WebSphere Application Server decodes it internally, as usual. If a custom provider is used, WebSphere Application Server passes the binary information to the user-defined provider if one is available. If one is not available, an error occurs. It is then the responsibility of the provider to return to WebSphere Application Server a clear text password that is represented by that binary data.

This model of lazy read/write means that at any time in a WebSphere Application Server cell, passwords might be encoded by using the default mechanism, and other passwords might be encoded or encrypted by using a custom mechanism. And, if the provider allows for updating the encryption key (as ours does), multiple passwords might be stored within the same cell that are encrypted by using different keys. Thus, the provider must be able to decrypt a password encrypted by using any previous encryption key. Our provider addresses this by keeping multiple key versions and embedding the key version in the information about the encrypted password.
Our provider design highlights

The remainder of this article documents the IBM Hybrid Cloud Services custom password encryption provider. The code is not freely available, nor it is available for download with this article. Rather, this asset is available only by engaging with IBM Hybrid Cloud Services in a billable services engagement. During this engagement, an IBM Hybrid Cloud Services consultant works with you to understand and document your requirements and tailor the asset if needed to better meet your use cases. They provide you with your own copy of the asset code, which you can then use in your enterprise. The goal in making this code available as an IBM Hybrid Cloud Service is in enabling you to pass a security audit where this issue is raised.

The key design decision in the IBM Hybrid Cloud Services custom password encryption provider implementation was to use the existing Java Cryptography Extension (JCE) infrastructure as much as possible. Therefore, the symmetric keys that are used for password encryption and decryption are stored by using the Java KeyStore support. We specifically chose to use the JCEKS format because this format properly supports symmetric keys. Java KeyStores allow for the keys stored in the keystore to be encrypted by using a password, and they allow for the keystore file to also be encrypted by using a password. Rather than asking a human to come up with an appropriate password, we automatically generate a long (30 or more characters) alphanumeric password and stash it in a second file – our stash or master password file. To prevent trivial snooping, that file’s contents are XOR’ed (by using a four character XOR mask) to make guessing difficult.

If you are using the file-based repository in the Federated Repositories user registry, user passwords are one-way salted hashed passwords. WebSphere Application Server does not use this SPI for writing those user passwords.

When WebSphere Application Server needs to encrypt any system or WebSphere Application Server password, the provider is called. The provider is expected to return a string and encrypted bytes. When WebSphere Application Server needs to decrypt a password, it checks to see whether the custom provider or default provider was used. If the custom provider was used, it is given a string (defined by the provider) and bytes. The provider’s job is to return the password as a char array. We take advantage of that string to store meaningful information.

WebSphere Application Server does not automatically encrypt all passwords. It starts only by using the code for new encryptions, so our code added functions to force WebSphere Application Server to reencrypt all passwords.

To enable managing and periodically updating the encryption keys without having to stop the entire cell, we need to support multiple encryption key versions at the same time. To do this, in the string that is given to WebSphere Application Server, we embed the key version that is used to encrypt that password. The EncryptionKeyManager class maintains a set of encryption keys (ordered by creation time). Whenever a string is decrypted, the key that is used to encrypt is used (assuming it is still available). When a string is encrypted, the most recent key is used. When servers are started, they read into memory the current version of the keystore and, therefore, all of the current keys. If a decryption fails because of a missing key version, the CustomPasswordEncryptImpl class automatically reloads the keystore to detect if a newer key is available. This might happen
if a new encryption key is added (and the DMgr is restarted), but the application servers are not restarted. Therefore, our solution does not require application server restart after the initial configuration to effect changes to the keys used for encryption.

Strong cryptography is obviously a key requirement of this solution. Therefore, we made the following decisions about cryptography usage:

- We use the Java JCEKS keystore provider, because it can store symmetric keys.
- We use a Java JCE Cipher that uses AES encryption with Cipher Feedback Mode. We also use no padding to avoid adding characters to passwords. Therefore, the precise provider string is AES/CFB/NoPadding.
- Because we are using a CFB algorithm, we must also store the initialization vector that is used for encrypting data. We allow the JCE run time to create a random initialization vector each time a Cipher is used to encrypt a password. We then store that vector with the password for initializing the Cipher before decryption. Using a separate vector for each password limits its reuse and improves the security of the encryption. Even if the same password is encrypted by using the same key, the resulting encrypted data is dramatically different.
- We use 128-bit key lengths for the AES encryption. Longer key lengths require special JDK JCE jurisdiction files. These special files are limited by US Federal export laws.
- We add random salt to the start of each password before encryption to further randomize the information that is stored, making decryption even more difficult.

We chose not to make the keystore, algorithm, or encryption type configurable. Doing so can significantly complicate the implementation and raise the possibility that someone might accidentally configure very weak password storage. Because all configuration information is managed by a single class, anyone with source code access (and appropriate cryptographic skills) can change the code to use other algorithms or configurations.

One obvious question to consider when you add support for the encryption of passwords is what impact it might have on performance. Cryptography is expensive, and performing cryptographic operations can have a serious impact on the performance of applications. Fortunately, WebSphere Application Server aggressively caches (in memory) decoded/decrypted passwords. Therefore, runtime performance should not be effected by this solution. However, there is a slight performance overhead when you start the servers (while they read the passwords) and make administrative changes that impact passwords.

A final concern pertains to the keeping of passwords in memory. Unfortunately, while the concern is valid, there is no feasible way around this issue. As required by the Java EE Connector Architecture (J2C) specification, every request to a resource must first check with a login module to obtain credentials for the back-end resource. The login module must provide, among other things, the cleartext password so that the underlying resource can use it (applies only to resources that need passwords). If passwords are not remembered in decrypted memory, every call to getConnection() on every resource from every application results in a password decryption, which is likely to significantly impact performance. Because getting passwords from a memory image of a process is not easy, we consider this risk acceptable given the real performance benefit.
Security implications
Before the use of this enhancement, an attacker had to exfiltrate a password-containing file and then determine how to undo the XOR and base64 encoding. With this feature, an attacker must exfiltrate the password-containing file, the encryption key files, and the JAR file that contains the IBM Hybrid Cloud Services asset. (The encryption key files must be on the machine and have the same read access permissions as the password file.) Then, the attacker needs to use these files together to decrypt the passwords.

The value of this solution is in the cases where the WebSphere Application Server XML configuration files are shared with third parties. Storing the passwords in an encrypted form prevents those third parties from determining the real value of passwords that are embedded in the XML configuration files. This presumes, of course, the third party is not provided a copy of the encryption keys, which is why this solution stores the encryption keys outside of the XML configuration tree.

When you back up the system, consider the fact that the backups contain both the encrypted passwords and the decryption keys, and the JAR file is used for decryption. For maximum security, carefully protect those backups, and consider storing the decryption keys on a separate backup. If you choose the latter option, be careful not to lose those keys. The passwords cannot be recovered without the decryption keys!

Availability implications
The decision to encrypt passwords by using secret keys leaves open the possibility of a total and unrecoverable cell outage if an encryption key is lost. Therefore, you must back up your cell before any key management operations, and use great care when you perform any key management activity. If an action results in the loss of encryption information that is still in use, there is no way to recover it without backups.

Managing the keys: The passmgr utility
A simple command line utility (passmgr.bat/sh) is provided to manage the encryption data. The launch script has UNIX® and Microsoft® Windows® versions. The passmgr script follows the standard documented thin client launch approach.

After you copy passmgr.bat/sh to some directory and edit the script to specify the correct value for WASHOME, the tool is launched by using passmgr.bat/sh, for example:

passmgr.bat

In most cases, you want to provide the profile name of the WebSphere Application Server profile that the tool should work with. If no profile is specified, the default profile is used. To specify a profile, start the passmgr.bat utility with a profileName, for example:

passmgr.bat -profileName Dmgr01

After you specify the profile name, you can define several other optional arguments and the mandatory command. The following listing shows the commands that are available. If you try to run the tool without providing a command, you get an error message that indicates your options.
passmgr.bat
No arguments specified
Usage:  <command> [-dir <dir>] [-key <key#>] [-host <dmgr host>]
[-port <dmgr port>] [-trace]
where <command> is one of:
create - create keyfile
addkey - add a key to keyfile
info - print information about a keyfile
deletekey - delete a key from keyfile
scanall - scan/read all passwords (may aid in debugging app server issues
forceencrypt - force the reencryption of all existing passwords
changemasterkey - change the master key protecting the keyfile

The optional arguments have the following meanings:

• -dir. Specify an alternative root for the keyfiles, which is useful if you want to create a keyfile,
but do not want to update the keyfiles that are used by the current profile. It is handy for
.testing.
• -key. For commands that require a key number (for example, deletekey), this is the key
number.
• -host. For those commands that talk to the dmgr (for example, scanall and forceencrypt), you
can specify a non-default dmgr hostname. The default is localhost.
• -port. For those commands that talk to the dmgr (for example, scanall and forceencrypt), you
can specify a non-default dmgr port. The default is 8879.
• -trace. Output trace information to the console to indicate what the management tool is doing.

Conclusion

IBM WebSphere Application Server (and products that are built on top of WebSphere Application
Server) stores system account passwords in various files within the WebSphere Application Server
profiles file structure. These passwords are encoded, not encrypted. Encrypting these passwords
can increase security for a certain class of vulnerability. It is debatable whether such encryption of
system passwords is required by the PCI-DSS standard or other standards, but IBM is frequently
asked to support password encryption.

WebSphere Application Server provides an SPI for customizing the processing of these system
passwords. This article described an IBM Hybrid Cloud Services custom implementation of
this SPI. If you are interested in deploying this custom implementation or learning more about
WebSphere Application Server security, contact IBM Hybrid Cloud Services to engage with a
security consultant.

Acknowledgments

I thank my colleagues Paul Ionescu, Bill O'Donnell, Tom Alcott, Simon Kapadia, T. Rob Wyatt, Pete
Neergaard, David Mundhenck and Paul Glezen for their valuable input and assistance.

I especially thank Keys Botzum who wrote the original asset, and original drafts of a significant
portion of the text in this article.
Related topics

- IBM Cloud Professional Services
- WebSphere Application Server security
- Advanced Authentication in WebSphere Application Server
- Advanced security hardening in WebSphere Application Server V7, V8 and V8.5, Part 1: Overview and approach to security hardening
- WASDev Developer Center
- Enabling custom password encryption
- Block cipher modes of operation (cryptography)
- Payment Card Industry Data Security Standard: Document library
- Encrypting passwords in config files – secure or not?