# Audit Report: CASA 1.6

March 6, 2006

| | |
|---|---|
| Package: | CASA |
| Version: | 1.6.265 |
| Product: | SUSE LINUX 10.1 |
| Reason: | - request for inclusion in SUSE Linux 10.1<br>- sensitive security component (manages user credentials) |
| Scope: | - crypto: password generation<br>- crypto: encryption of secrets (persistent store and cache)<br>- crypto: possible leaking of credentials<br>- design: access control to secret store<br>- design: general source-code review (C and C# only) |
| Reviewer: | Thomas Biege <thomas@suse.de> |
| Review-Version: | 0.1 |
| Bug-ID: | 132707 |
| References: | none |

# 1 Introduction

The Common Authentication Service Adapter (CASA) manages credentials between *eDirectory*, *FireFox*, *Mozilla*, *KDE Wallet*, and *Gnome Keyring*.

# 2 Audit Environment

- SUSE LINUX 10.0 x86
- CASA 1.6.265
- programming language: C#, C, Java

# 3 Methodology

Due to the complextity of the code and due to the limited time available the auditor was forced to focus on the most sensitive parts. A line-by-line source-code review as well as stress-testing during runtime was not made but should be done later. Additionally errors found in the included test-code were irgnored.

# 4 Code Review - Crypto

## 4.1 File: c_micasad/lss/Rfc2898DereiveBytes.c

- **medium**: Rfc2898DeriveBytes: line 117: The salt is generated from the hash-code of the password. This procedure neutralizes the effect of a salt for key generation. The reason for a salt is to make a codebook attack impractical by generating $2^n$ (where n is the length of the salt in bit) keys from just one password, therefore the salt has to be random and changing [1]. If the salt is derivated from the password it can be considered constant because the KDF (Key Derivation Function) always generates the same key for a password. The pseudo random number generator should be seeded with a random quantum from `/dev/urandom` or at least accumulated environmental values (i. e. time, PID etc. is not enough because it may not change between successive calls).

## 4.2 File: c_micasad/cache/KeyValue.c

- **medium**: XORValue: line 199: Key-value pairs are managed in the cache. The value is encrypted using XOR and C#'s PRNG `Random()` which is seeded with the key creation time in ticks (1 tick = 100 ns). Timers do not provide much entropy, depending on the resolution only n bits are changing often enough. Assuming a runtime of 12 hours at maximum an attacker needs about $2^{39}$ guesses

to crack **every** password in the cache. Another problem arises from a situation were two or more secrets are encrypted within one tick. The results can be XOR'ed to remove the "key" and start a statistical attack on the XOR'ed plaintext. This kind of obfuscation technique is commonly used in practice but it should be improved by also using real secrets (like the *Master Password*, *Master Passcode*, or *Desktop Secret*) and a counter additionally to the timestamp. This is also very important because the XOR-encryption is used for the system password of an user too, therefore the protection of the password (which normally is a hash-digest from DES or from a real hash-algorithm) is reduced.

## 4.3 File: c_micasad/lss/LocalStorage.cs

- low: various places: Files moved and deleted should be checked for ownership. If a user ownes a file with the same name it gets destroyed. Fortunately it is not possible to remove files of other users because the `File` methods always operate on the link name and not on the file the link points to. And additionally, `Move()` throws an exception if the destination already exists.

## 4.4 File: c_micasad/lss/CASACrypto.cs

- **medium**: various places: The *initialization vector* (IV) is not used correctly when calling `RijndaelManaged.CreateEncryptor()` or when it is copied in a for–loop. The IV is the same as the key which makes the IV constant in relation to the key. This practice leaks information about the first plaintext block. See A1 for more information.

- **medium**: DecryptMasterPasscodeUsingString: line 434: The method `FileStream()` will follow links. This can be used to access files of other users[1]. Possibly a dictionary attack on the other users file could be executed to gain access.

- low: GenerateMasterPasscodeUsingString: line 532: Using a constant value that is encrypted with the MPC is not a good idea. It leaks information to an attacker. Better use a hash–digest.

- low-medium: ValidateMasterPasscodeUsingString: line 561: Using a constant value to validate the MPC is not a good idea. This infomation can be used by an attacker to execute a dictionary or brute–force attack on the key in a too easy way. Better use a hash–digest to double the effort needed. Or copy the technique used in [1] for the KDF.

- low-medium: various places: It can be too late to use method `Mono.Unix.Native.Syscall.chmod()` here because an local attacker may already have aquired a

[1] There will only be an exception thrown if the privileges do not suffice.

file descriptor of the file if the permission are too lax while using `FileStream()`. Instead the system call `umask(2)` should be used very early in the daemon's code.

### 4.5 Notes

- Memory that contains secrets should be explicitly erased by overwriting the content with zero (or a like).
- Memory that contains secrets should be protected against swapping by using `mlock(2)`.

## 5 Code Review - Design

### 5.1 File: c_micasad/communication/UnixCommunication.cs

- low: UnixCommunication: line 55-60: The unix-domain socket is deleted and created later. If a local user creates his/her own socket named `/tmp/.novellCASA` inbetween the code terminates with an exception when using `Bind()`.

### 5.2 File: c_micasad/lib/communication/UnixIPCClientChannel.cs

- **HIGH**: Open: line 46: The client opens the unix-domain socket without checking the UID of the process listening on the other side. This leads to a serious problem that can be exploited by a local user to fake the `miCASA` service and steal the master password from the user. This attack is eased due to the fact that `miCASA` always deletes the socket from the filesystem when it starts up. Between deletion on creation of the socket exists the possibility for an attacker to create the socket of his/her own even if it exists before and belongs to root. (see 5.1)

### 5.3 File: c_micasad/lib/communication/UnixIPCClientChannel.cs

- low: Read: line 97: The integer `msgLen` is set by reading a value from the socket. This source can be considered untrusted and should not be used without further verification. For example a buffer based on this value was allocated later. This could lead to a denial- of-service vulnerability when exception `System.OutOfMemoryException` is triggered. This exception is catched and `null` is returned. This only affects clients using the socket.

## 5.4 File: c_micasad/lss/CASACrypto.cs

- low: ReadFileAndDecryptData: line 235: Check if `fsDecrypt.Length` is lesser then 32. This results in a negative number. Additionally it is better to use an unsigned integer.
- low: ReadFileAndDecryptData: line 236: Check for an upper limit of `fileLen`. Files are often much bigger then the available RAM which will result in a memory exhaustion.

## 5.5 File: c_micasad/verbs/ObjectSerialization.cs

- low: ProcessRequest: line 90,110: The value of `inMsgLen` should be checked to be greater then 6 and below an upper bound for memory allocation. Otherwise all available memory might get allocated which results in a denial-of-service situation.

## 5.6 File: c_micasad/verbs/OpenSecretStore.cs

- low: ProcessRequest: line 85,86: The value of `ssNameLen` should be checked to be below an upper bound for memory allocation. Otherwise all available memory might get allocated which results in a denial-of-service situation.
- note: ProcessRequest: line 114: Shouldn't `msgId` be 0x1001?

## 5.7 File: c_micasad/verbs/SetMasterPasscode.cs

- low: ProcessRequest: line 82: The value of `passcodeLen` should be checked to be below an upper bound for memory allocation. Otherwise all available memory might get allocated which results in a denial-of-service situation.

## 5.8 File: c_micasad/common/SessionManager.cs

- note: CheckAndDestroySession: line 281: What `$PATH` environment varaiable is used here?
- note: CheckAndDestroySession: line 296: The while-loop seems useless because it iterates only once.

## 5.9 File: c_micasadk/sscs_ndk.c

- **medium-high**: various places: All calls to `sscs_Utf8Strlen()` should make sure that their arument is null-termiated.

- **medium-high**: various places: All calls to `sscs_Utf8Strcat()` should make sure that the destination buffer is big enough. And that the resulting string gets null-terminated.
- **medium-high**: various places: All calls to `sscs_Utf8Strcpy()` should make sure that the destination buffer is big enough. And that the resulting string gets null-terminated.
- **medium-high**: various places: All calls to `memcpy()` should make sure that the destination buffer is big enough.
- **HIGH**: sscsshs_ChkEscapeString: line 242: The for-loop uses `len = sscs_Utf8Strlen (entryBuf) + 1` as limit for escaping chars in `entryBuf`. The result will be written to `tempBuf` which is of size `NSSCS_MAX_SECRET_BUF_LEN - SSCS_CRED_SET_LEN`. This may lead to a possible overflow on the heap. Please check if `len` is bigger then the size of `tempBuf`. At the end of this function `sscs_Utf8Strcpy()` is used to copy the escaped string in `tempBuf` to `entryBuf`. This may lead to another overflow. There may be an internal limit that is assumed for these buffer, I did not check for it.

## 5.10 File: auth_token/krb5_token/linux/get.c

- low: Krb5AuthTokenIf_GetAuthTokenCredential: line 214,215: A one–byte buffer overflow can happen here. The length parameters `gssDisplayName.length` and `encodedTokenLen` are checked some lines before but in `memcpy()` 1 is added. I am not sure how much influence an attacker can take here. Furthermore always make sure that strings get null–terminated. This is not assured in other places too. I did not show them here because it happens during parsing the config file which doesn't matter with regards to security.

## 5.11 File: c_micasacache/sscs_ipc.c

- low: ipc_unx_read: line 185: The return value 0 (End-Of-File) should be handled to avoid looping forever.

## 5.12 File: c_micasacache/sscs_unx_ipc_client.c

- low-medium: Tokenize: line 127,132,149,154: How is it assured that `sscs_Utf8Strcpy()` will not copy more bytes then memory available?
- note: various places (memcpy): Instead of using several dozens calls to `memcpy()` it may be more effective to lay a struct over a byte buffer.
- low: various places (`bufLen`): The value of `bufLen()` should be checked before it is used for allocating memory to avoid a denial-of-service.

### 5.13 File: c_adlib/ad_gk/native.c

- low-medium: various places: Does the destination buffer fit while using `strcpy(3)`? Often it is not assured that the string gets null-terminated.

### 5.14 Notes

- **important** The methods using the `.miCASA` files created/used in the user's home directory all follow symbolic links. This can be used to create arbitrary files in arbitrary locations on the filesystem with root privileges.
- File descriptors should be protected by setting the *Close on Exec* flag by using the function `fcntl(2)`. This is especially important for client applications that hold a file descriptor for the unix-domain socket and already have passed authentication.
- In file `c_micasad/init/Main.cs` the method `CheckIfMiCASAdIsRunning` use a uncommon method to detect if an instance of the daemon is already running. Common practice is to create a PID file and send signal 0 for testing.
- Values from untrusted sources should not be used directly - for example as parameter of `new` or in pointer arithmetic. Instead their value-range and type should be checked. Some examples are mentioned above to show some obvious denial-of-service conditions that can be triggered by a malicious user.
- It is a risk to act as root in user's home directory. Beside the security risk it is not nice to have root–owned files in a directory that another entity ownes.

## 6 Status

Review finished. What is left?

- policy code
- network communication (is there any?)
- possible "serialization attack" via unix-domain socket
- Java source

## 7 Appendices

### 7.1 A1: How to use an Initialization Vector

There a various ways to generate an IV. But before we get into the details here let's see why an IV was used. To randomize the encryption of the same plaintext block to hide

patterns (like it happens in the *electronic code book* (ECB) mode) the plaintext block $P_i$ is XOR'ed with the ciphertext block $C_{i-1}$. This mode is called *cipher block chaining* (CBC). The first plaintext block $P_0$ has no preceding ciphertext block, therefore the IV is used as reparation. To fulfill the task of hiding patterns the IV should **not be constant**, should **not be a simple counter**, should **not repeat** (also take care of integer wrap arounds) but can be public. This goal can be achieved by generating a random IV or by using a *nonce* (*n*umber used *once*) IV. The nonce IV is easier to handle in practice.

The class `RijndaelManaged` has a method called `GenerateIV()` which can be used. The IV can be stored in plaintext along the encrypted data (maybe like: [IV][encryped message]).

# References

[1] B. Kaliski; PKCS #5: Password-Based Cryptography Specification Version 2.0; RFC2898