Heap Inspection (CWE-244)

Vulnerability Severity: Low

Impact of Exploitation: Unauthorized Access to Sensitive Information, Privacy Violation

File Name	Line Number
COTURN	
\coturn\src\apps\relay\dbdrivers\dbd_mysql.c	46, 55
\coturn\src\apps\relay\dbdrivers\dbd_redis.c	52
\coturn\src\apps\relay\ns_ioalib_engine_impl.c	3567

Applications' variables/objects stored in memory without encryption can be accessed with unauthorized access by malicious users. For instance, a "authrized" attacker can attach a debugger to a running process or obtain a memory dump of the process from a swapfile or crash dump file. Using these methods, an attacker can discover user passwords in memory and use this information on the system to easily impersonate a user profile.

String variables/objects are immutable, meaning once a string variable/object is defined, its value cannot be changed or deleted. Therefore, these strings remain in memory for an uncertain period - possibly in multiple locations - until they are deleted by the garbage collector. Sensitive data defined with these strings (e.g., passwords, encryption keys, API keys, etc.) are displayed in plaintext in memory throughout their lifecycle in an uncontrolled manner. This exposes the risk of attackers with memory access privileges viewing and capturing sensitive data in plaintext. When applications position sensitive data in memory in plaintext, it is referred to as a heap inspection vulnerability.

To illustrate the heap inspection vulnerability concretely, vulnerable code blocks and secure alternatives are provided.

C++ Unsafe Code Block:

```
char* password = (char*) malloc(256);                    // UNSAFE
int i=0;
char ch;
ssize_t k;
while(k = read(STDIN_FILENO, &ch, 1) > 0)
{
    if ((ch == '\n') || (i >= 255))
    {
        password[i]='\0';
        break;
    }
    else {
        password[i++]=ch;
    }
}

// ..
// Verify password
// ..

free(password);
```

C++ Safe Code Block:

```
volatile char* password = (char*) malloc(256);          // SAFE
int i=0;
char ch;
ssize_t k;
while(k = read(STDIN_FILENO, &ch, 1) > 0)
{
  if ((ch == '\n') || (i >= 255))
  {
    password[i]='\0';
    break;
  }
  else {
    password[i++]=ch;
    fflush(stdin); // Zero-out input stream
  }
}

i=0; // Zero-out password length
// ..
// Verify password
// ..

while (i <= 255) {
  password[i++] = 0; // Zero-out password, clearing it from the
heap
}
free((void*)password);
```

As seen in the C++ example, sensitive data (in this case, password data) is defined as an immutable object. Sensitive data should be defined as mutable. For this purpose, objects that will hold sensitive data in C++ applications can be defined as volatile. This will close the loophole in this way.

COTURN:

\coturn\src\apps\relay\dbdrivers\dbd_mysql.c

```
#include "../mainrelay.h"

#if !defined(TURN_NO_MYSQL)
#include <mysql.h>

////////////////////////////////////////////////////////////////////////////////

static int donot_print_connection_success = 0;

struct _Myconninfo {
  char *host;
  char *dbname;
  char *user;
  char *password;
  unsigned int port;
  unsigned int connect_timeout;
  unsigned int read_timeout;
  /* SSL ==>> */
  char *key;
  char *ca;
  char *cert;
  char *capath;
  char *cipher;
  /* <<== SSL : see http://dev.mysql.com/doc/refman/5.0/en/mysql-ssl-set.html */
};
```

Şekil 1. Lack of Heap Inspection

Prevention of Vulnerability:

The recommended security measures to be implemented are generally as follows:

- Avoid storing sensitive data (e.g., passwords, encryption keys, API keys, etc.) in plaintext (unencrypted) form in memory, even for a short period.
- Prefer the use of specialized classes that store encrypted data in memory, making it more difficult to extract.
- When sensitive data needs to be used in its raw form, temporarily store it in a mutable (changeable) data type (e.g., byte arrays) to reduce readability if read from memory. Immediately after use, overwrite the memory locations of these data to minimize their exposure while in memory.
- Even when the above measures are implemented, ensure that memory dumps are not exchanged with untrusted parties. Reverse engineering of encrypted variables/objects or extracting and recompiling sensitive data bytes from memory may still be possible.

Even if a cleaned mutable data type is used instead of immutable as a precaution against Heap Inspection, or if sensitive data is fetched from memory by decrypting it with a decryption key, it is still possible to retrieve data from memory as a result of heap inspection attacks. However, implementing a sensitive data layering measure as described will significantly increase the amount of effort attackers need to exert. By keeping the security bar high to reduce the retrieval, display, and quantity of sensitive data in memory, the success of attackers in obtaining valuable data can be substantially diminished.

When avoiding Heap Inspection vulnerabilities, it is important to emphasize the following: Disclosing sensitive data to attackers is always possible when access to the memory dump or memory of an application is granted. The security measures mentioned serve as part of the defense-in-depth principle in protecting sensitive data when read access to memory is successfully obtained. These recommendations result in a significant reduction in the exposure and lifespan of sensitive data in

memory. However, it is a fact that, given sufficient time, effort, and unrestricted access to memory, the protection of sensitive data used by an application can only go so far.

The only way to overcome the Heap Inspection vulnerability is to reduce and minimize the display of sensitive data in memory and obscure this sensitive data throughout the entire memory.

References:

1. https://cwe.mitre.org/data/definitions/244.html