



ADITYA K. SOOD

Hacking IM Encryption Flaws

Difficulty



This paper sheds a light on encryption problems in Instant Messaging client's primary memory which lead to hacking. The IM clients have been used extensively all over the world to exchange messages between different parties.

Some of the clients are commercial and some of them are open source. But it has been noticed there are several issues of insecurity adhere to these clients. This includes unencrypted passwords in memory, Denial of service due to crashing, etc which are very common to these clients.

The configuration files leverage bundle of information of the IM clients running on the client systems. This is static behavior of IM clients to use configuration files. We will be talking in detail about the encryption problems in memory due to which password float in clear text in memory.

Encryption Stringency in IM clients

It has been noticed that number of Instant Messaging Clients does not encrypt passwords in memory. The username and password used by client to log in to centralized server for instant chatting somewhat remain in clear text in memory.

The primary memory of the running process of instant messaging client possesses the user credentials in clear text which is considered to be as vulnerability. This paper revolves around this specific problem of encryption pertaining to Instant Messaging clients.

As the credentials remain in clear text in memory it becomes possible to dump the content of that process in a raw format. Once the dump is extracted it is quite easy to find the username and

password. It is a potential threat or weakness from view point of client side security.

Even if the system is compromised by less authorized users with low privileges still it is easy to dump the memory and find the required credentials. So what is the real problem that leads to this kind of vulnerabilities?

Most of the Instant Messaging clients store user name and password in the process memory which is required for definite functioning of messaging clients. It depends a lot on the development team regarding the mechanism followed to encrypt or decrypt the passwords in memory or there is another feature to follow to make the encryption possible in memory. Encrypting passwords and stored as key in the memory.

This is one of the good practices to follow. For Example – Google Talk client encrypts the password and stored it in a key called *pw*. This key resides in the memory but it is very hard to find in the raw dump.

Similarly a reverse procedure is defined to decrypt it while comparing credentials with server database. Looking at this layout, it is defined that a well structured mechanism is to be designed for encrypting passwords in memory.

On the other hands this is necessity too. But unfortunately it is not the story of every client like Google Talk. We will dissect this vulnerability by analyzing raw dumps for certain client to see and check the flaw.

WHAT YOU WILL LEARN...

Working internals of Instant Messaging will be useful

Knowledge of Hashing Algorithms will prove beneficial

Cryptography concepts will be beneficial

WHAT YOU SHOULD KNOW...

The critical vulnerability of Client side Password Disclosure in Instant Messengers

The encryption flaw in password storage

Conducting memory test on live processes

Number of Instant message clients lack encryption mechanism to store passwords in memory.

This is a serious flaw from security point of view. What is the actual cause of this? The reasons are presented as below:

- Most of the clients store password in clear text. It has been noticed after the storage process the credentials are encrypted and compared with the required stored credential on the server side. This is flaw oriented process because the encryption procedure is implemented after the password is present in clear text. It is not considered to be as a good approach because it results in leakage of credentials in process memory.
- The second reason is there is no hashing procedure is followed. The hashing is one of the best approaches which need to be followed. But this is not so. The IM clients lack this. There is no hashing mechanism is followed or implemented. This is very fruitful from security realm if password is stored as a hash key in the memory. The hashing algorithm generates the same hash every time when a specific string is passed to it. Due to this reason it becomes easy to compare the hashes directly with the stored hash on the server side and there is no need to compare the passwords in clear text. The comparison of credentials is done through hashing not by simple text. For Example:- MD5 hashing algorithm can be used to hash the password. Another MD5 hash for same string can be stored on server and comparison can be done. As MD5 is based on One way function as a result in memory dumps it is somewhat a hard task to accomplish. SHA-1 can also be used. Preferably any standard hashing algorithm is used to complete this task.
- It has been analyzed that no salt generation is done even when hashing procedure is followed. Salt is a string of random numbers which is used altogether with password and appended

in front. After this the hash is computed. This process of salt generation and implementation makes the storage and comparison of IM credentials more strong. This no doubt hardens the process of encryption .Basically salt are used to dethrone the direct dictionary attacks on the hashes. On the contrary it is a good mechanism to follow in IM client password storage. But incessantly the IM client does not use this.

These are the critical issues which IM lacks which leads to hacking of passwords in memory.

Firstly we will analyze a simple working algorithm of hashing passwords and salt generation.

Let's have a look at implementation of hashing algorithm in ruby.

A code snippet (you can see this in Listing 1). So that's how hashing is implemented.

Listing 1. Salt implementation with SHA

```
require 'digest/sha2'

# This module contains functions for hashing and storing passwords module Password

# Generates a new salt and rehashes the password
def Password.update(password)
  salt = self.salt
  hash = self.hash(password,salt)
  self.store(hash, salt)
end

# Checks the password against the stored password
def Password.check(password, store)
  hash = self.get_hash(store)
  salt = self.get_salt(store)
  if self.hash(password,salt) == hash
    true
  else
    false
  end
end

# Generates a psuedo-random 64 character string
def Password.salt
  salt = ..
  64.times { salt << (i = Kernel.rand(62); i += ((i < 10) ? 48 : ((i < 36) ?
    55 : 61 ))) .chr }
  salt
end

# Generates a 128 character hash
def Password.hash(password,salt)
  Digest::SHA512.hexdigest("#{password}:#{salt}")
end

# Mixes the hash and salt together for storage
def Password.store(hash, salt)
  hash + salt
end

# Gets the hash from a stored password
def Password.get_hash(store)
  store[0..127]
end

# Gets the salt from a stored password
def Password.get_salt(store)
  store[128..192]
end
```

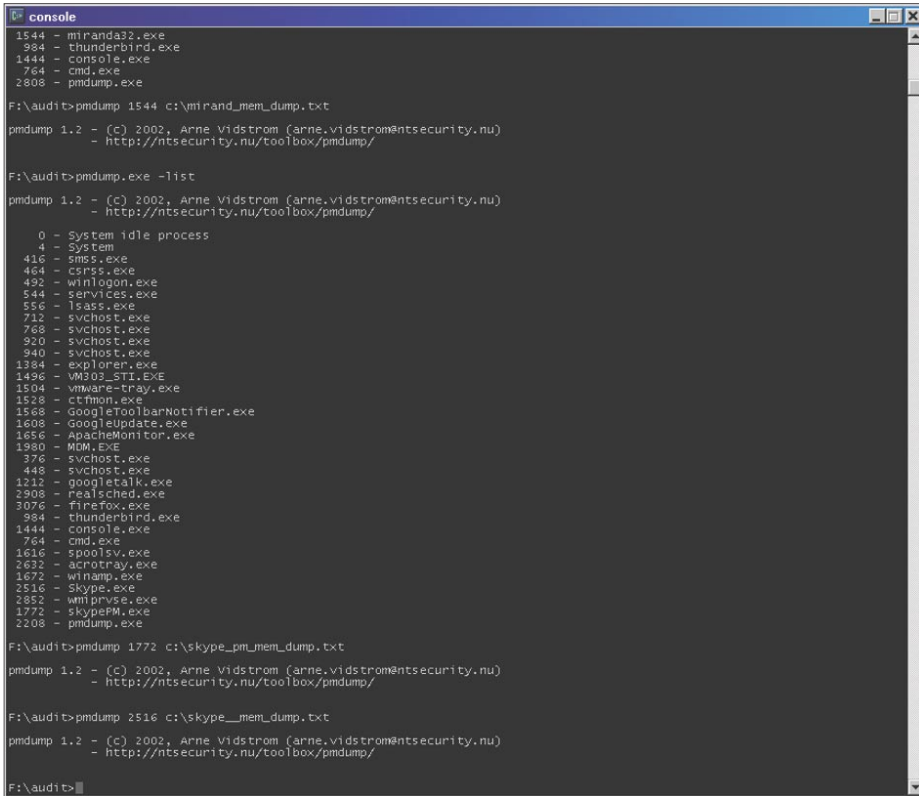


Figure 1. Process Memory Dumper in action



Figure 2. Skype Raw Memory Dump with traced username

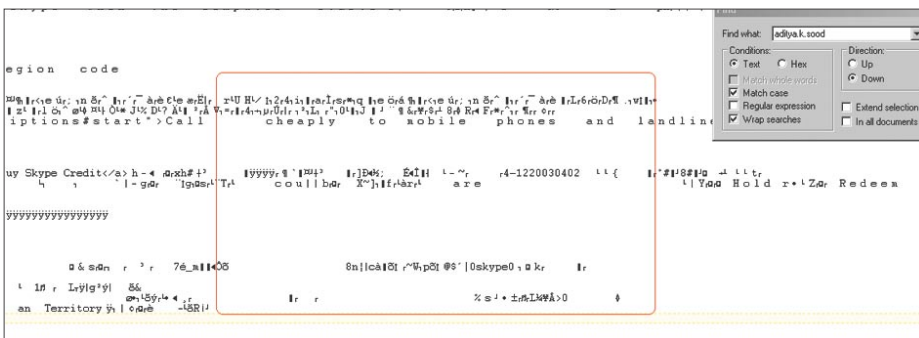


Figure 3. Skype Raw Memory Dump with traced password

Clear Text Credential Disclosure Vulnerability in SKYPE IM

In order to prove this flaw an example has been constructed from the vulnerability I have found in SKYPE Instant Messenger. A little test will be conducted to see whether the vulnerability is there or not. It has been found that SKYPE fails to encrypt the password properly.

Due to which password resides in clear text as per the problem discussed above. The credentials can be extracted in clear text by dumping process memory of the live skype process when a connection is set.

The vulnerability allows anyone with access to the client system to obtain the username and password.

Additionally, this vulnerability could also be exploited by fooling the user to execute malicious code which would dump the memory of the process skype.exe. The skype uses skype.exe and skypeppm.exe processes while communicating.

Description

A test account is created with username skypeimtest and password 0skype0. Live connection is set to the yahoo service. The process is dumped and analyzed to prove the concept.

- Step 1: Dumping memory with pmdump utility (see Figure 1)
- The pidgin memory dump is extracted to a txt file for analysis.
- Step 2: Analyzing Dumps
- The analysis shows the skypeimtest user account (see Figure 2),
- The username can be seen in clear text,
- The password 0skype0 is appeared (see Figure 3),
- The password can be seen in clear text. This vulnerability proves that encryption mechanism fails to encrypt the password of client in the process memory. The only stringency is sometimes it is hard to search clear text in this bunch of raw data. But there is always a way to do it. That hacker knows.

Listing 2. Linus Security Module (LSM) - Part 1

```

#include <linux/config.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/security.h>
#include <linux/file.h>
#include <linux/mm.h>
#include <linux/mman.h>
#include <linux/pagemap.h>
#include <linux/swap.h>
#include <linux/smp_lock.h>
#include <linux/skbuff.h>
#include <linux/netlink.h>
#include <linux/ptrace.h>
#include <linux/sysctl.h>
#include <linux/moduleparam.h>
+
+#define RT_LSM "Realtime LSM " /* syslog
+           module name prefix */
+#define RT_ERR "Realtime: " /* syslog error
+           message prefix */
+
+
#include <linux/vermagic.h>
MODULE_INFO(vermagic, VERMAGIC_STRING);
+
+/* module parameters
+ *
+ * These values could change at any time due to some process
+           writing
+ * a new value in /sys/module/realtime/parameters. This is OK,
+ * because each is referenced only once in each function call.
+ * Nothing depends on parameters having the same value every
+           time.
+ */
+
+/* if TRUE, any process is realtime */
+static int rt_any;
+module_param_named(any, rt_any, int, 0644);
+MODULE_PARM_DESC(any, " grant realtime privileges to any
+           process.");
+
+/* realtime group id, or NO_GROUP */
+static int rt_gid = -1;
+module_param_named(gid, rt_gid, int, 0644);
+MODULE_PARM_DESC(gid, " the group ID with access to realtime
+           privileges.");
+
+/* enable mlock() privileges */
+static int rt_mlock = 1;
+module_param_named(mlock, rt_mlock, int, 0644);
+MODULE_PARM_DESC(mlock, " enable memory locking privileges.");
+
+/* helper function for testing group membership */
+static inline int gid_ok(int gid)
+{
+   if (gid == -1)
+       return 0;
+
+   if (gid == current->gid)
+       return 1;
+
+   return in_egroup_p(gid);
+}
+
+static void realtime_bprm_apply_creds(struct linux_binprm *bprm,
+           int unsafe)
+{
+   +
+   +           cap_bprm_apply_creds(bprm, unsafe);
+   +
+   +           /* If a non-zero 'any' parameter was specified, we
+   +           grant
+   +           * realtime privileges to every process. If the
+   +           'gid'
+   +           * parameter was specified and it matches the group id
+   +           of the
+   +           * executable, of the current process or any
+   +           supplementary
+   +           * groups, we grant realtime capabilities.
+   +           */
+   +
+   +           if (rt_any || gid_ok(rt_gid)) {
+   +               cap_raise(current->cap_effective, CAP_SYS_
+   +                   NICE);
+   +               if (rt_mlock) {
+   +                   cap_raise(current->cap_
+   +                       effective, CAP_IPC_LOCK);
+   +                   cap_raise(current->cap_
+   +                       effective, CAP_SYS_RESOURCE);
+   +               }
+   +           }
+   +
+   +           }
+   +
+   +static struct security_operations capability_ops = {
+   +   .ptrace = cap_ptrace,
+   +   .capget = cap_capget,
+   +   .capset_check = cap_
+   +       capset_check,
+   +   .capset_set = cap_
+   +       capset_set,
+   +   .capable = cap_capable,
+   +   .netlink_send = cap_
+   +       netlink_send,
+   +   .netlink_recv = cap_
+   +       netlink_recv,
+   +   .bprm_apply_creds = realtime_bprm_apply_
+   +       creds,
+   +   .bprm_set_security = cap_bprm_set_
+   +       security,
+   +   .bprm_secureexec = cap_bprm_secureexec,
+   +   .task_post_setuid = cap_task_post_
+   +       setuid,
+   +   .task_reparent_to_init = cap_task_reparent_
+   +       to_init,
+   +   .syslog = cap_syslog,
+   +   .vm_enough_memory = cap_vm_enough_memory,
+   +};
+   +
+   +#define MY_NAME __stringify(KBUILD_MODNAME)
+   +
+   +static int secondary; /* flag to keep track of how we were
+   +           registered */
+   +
+   +static int __init realtime_init(void)
+   +{
+   +   /* register ourselves with the security framework */
+   +   if (register_security(&capability_ops)) {
+   +
+   +       /* try registering with primary module */
+   +       if (mod_reg_security(MY_NAME, &capability_
+   +           ops)) {
+   +           printk(KERN_INFO RT_ERR
+   +               "Failure registering "
+   +               "capabilities with
+   +               primary security module.\n");
+   +           printk(KERN_INFO RT_ERR "Is

```

Risks posed by this vulnerability

- Extracting passwords from memory possesses serious risk because it compromises the credentials of the required user and the account associated with it. The user related information can be exposed to the hacker there by leveraging sensitive information pertaining to the user whose account is compromised. It depends on the user whether this

account is same for exchanging mails. If this is so then the risk factor is big because attack vector is diversified.

This favors the brute forcing attack as credentials are present in clear text. An attacker can launch brute force attacks successfully. It is possible for an attacker to construct a file of required clear text words and start the attack which is quite hard when passwords are stored in encrypted form.

- It also shows the design flaw in an application. Usually while designing an application lot of factors play role. The application is constructed by implementing procedures for number of objects. There is a element of interdependency between objects that are used. The working functionality of one object somehow depends on the other. If the functionality of one object is weak it definitely impacts the functionality of other object. Similarly if an application has weak methods it surely lowers the robustness of whole application there by affecting the stature of an application.
- The operating has complexity at lower level. If an application code is not designed properly and code optimization checks are not performed then it is possible to have cache of user supplied data somewhere in the process memory or disk space. The shared library working procedure should be traversed properly to compile and link code effectively.
- Well jumping on to automation it is possible to design memory retrieval tools as whole because certain procedures are required to complete the task generically. It means if an attacker understands the flow of IM application and process characteristics he can design his own tool to retrieve passwords from IM process memory.

On the 'Net

- <http://technet.microsoft.com/en-us/library/ms190730.aspx>
- <http://lwn.net/Articles/110346/>
- http://domino.research.ibm.com/comm/research_projects.nsf/pages/gsal.TCG.html

Listing 2. Linus Security Module (LSM) - Part 2

```
kernel configured "
+
+           "with CONFIG_SECURITY_CAPABILITIES=m?\n";
+           return -EINVAL;
+       }
+       secondary = 1;
+   }
+
+   if (rt_any)
+       printk(KERN_INFO RT_LSM
+              "initialized (all groups, mlock=%d)\n", rt_mlock);
+   else if (rt_gid == -1)
+       printk(KERN_INFO RT_LSM
+              "initialized (no groups, mlock=%d)\n", rt_mlock);
+   else
+       printk(KERN_INFO RT_LSM
+              "initialized (group %d, mlock=%d)\n", rt_gid, rt_mlock);
+
+   return 0;
+}
+
+static void __exit realtime_exit(void)
+{
+   /* remove ourselves from the security framework */
+   if (secondary) {
+       if (mod_unreg_security(MY_NAME, &capability_ops))
+           printk(KERN_INFO RT_ERR "Failure unregistering "
+                  "capabilities with primary module.\n");
+   } else if (unregister_security(&capability_ops)) {
+       printk(KERN_INFO RT_ERR
+              "Failure unregistering capabilities with the kernel\n");
+   }
+   printk(KERN_INFO "Realtime Capability LSM exiting\n");
+}
+
+late_initcall(realtime_init);
+module_exit(realtime_exit);
+
+MODULE_DESCRIPTION("Realtime Capabilities Security Module");
+MODULE_LICENSE("GPL");
```

We have listed some of the risks posed due to these types of encryption flaws in memory. Now we will look into the protection steps that are to be followed in order to combat against these attacks.

Protection Steps

- The very basic point is the type of security model followed while designing an application. It sets the design model in a way to impose security parameters on the object used in the application. The design model also suggests the way to secure the object access parameters in the memory through cryptographic models. It sets

an insight of secure software from over all perspective.

The second step is the use of encryption in a well structured manner even on client side. For actively working of software certain credentials are required every time to work dynamically. So the credentials need to be secured even on client side. Like it is stated above the skype issue. So the critical parameters should be encrypted in a potential manner which is not even visible in memory dumps. The possible solution is to generate a hash and it should be compared with the stored hash on server side.

Another good step is to assign security and access control parameters in uniquely manner while setting object in a software because if permissions are apply as a group it will result in weak security. If one object is compromised to some extent then there it is a possibility to use other object too with same security imposed as group. This is a good software design principle.

While applying cryptographic solutions strong algorithms must

be favored in order to increase the strength of software or application while coding it.

These are the very general solutions to follow and implement but a very good practice to follow.

Two Specific High End Solutions

These solutions are dependent on operating system too. The developer should use these features to avoid any vulnerable approach of dumping memory:

The technique of overwriting credentials in memory should be followed. As the password is not required it should be overwritten efficiently by using operating system libraries and internal API calls to shred the traces of password in the memory even when the application is dynamically active. The operating system code also handles password in memory so a proper approach of overwriting the sustained credentials will minimize the risk of stealing from physical memory.

- The second highly efficient technique is to lock memory pages to avoid memory dumps from the operating system. In windows you can set the parameter for locking pages to avoid dumps which is otherwise disabled by default. The user assignment folder in windows setting in group policy has parameter *Lock pages in memory* which will stop the dumping of physical memory. In Linux one can use LSM i.e Linux Security Module to configure the MLOCK i.e. memory lock. This is a standard code for LSM Module (see Listing 2).
- The use of hardware security modules i.e. HSM and Trusted Computing Architecture implements high end privacy but these are specific to CPU.

So that's how memory can be secured. We have found number of solutions to this. But if an attacker controlled the whole machine as root nothing works as such.

Conclusion

The memory encryption flaw leads to insecurity in an application or software. A proper design principle should be followed in a deeper manner to avoid inconsistency of this kind. Cryptographic solutions are required in this. The crypto functions should be implemented in a definite manner to drop down the vulnerable behavior on client side. It depends a lot on a developer in designing the working flow parameters in an application or software. A top to bottom, secure approach of software designing is required to combat against these flaws.

Aditya K. Sood

Aditya K. Sood is an independent Security Researcher and Founder of SecNiche Security. He is a Lead Author for Hakin9 group for writing security and hacking papers. His research has been featured in Usenix; login magazine and Elsevier Network Security Journals. Aditya's academic background holds a BE and MS in Cyber Law and Information Security from Indian Institute of Information Technology (IIIT-A). He had already spoken at conferences like EuSecWest, XCON, OWASP, CERT-IN etc. In addition to that He is a team lead at Evilfingers community. His other projects include Mlabs, CERA and TrioSec. He has written number of security papers released at packetstorm security, Linux security, infosecwriters, Xssed portal etc. He has also given number of security advisories to forefront companies. At present he is working as a Security Auditor in KPMG IT Advisory Services where he handles large scale security assessments project.

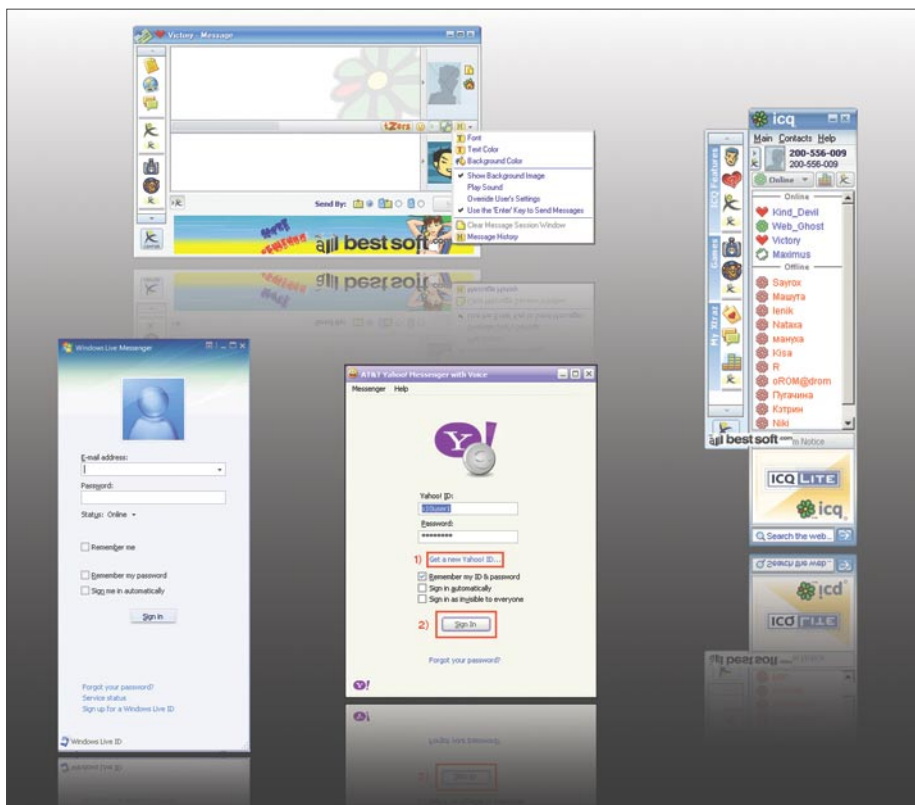


Figure 4. Messengers: ICQ, MSN, Victory