

Complex protection for secured messaging

Problem description

State of the art encryptions, especially ones based on perfect-forward security and end-to-end principles, are well-established algorithms, that take enormous amount of computational power to be broken. Methods of key-mutation make sure that it is technically and economically impossible to break any specific encrypted packet or packets stream, that have been captured on the transmission line or router, even if attacker exactly knows the software used to encrypt and send these packets.

But security level of communication is the security level of the most vulnerable part. Network communications were first an issue because they were very easy to capture. An attacker could connect to the transmission lines, that cannot be physically protected, especially in case of radio transmissions, or could capture packets on a router if it was poorly secured, or could use more “intellectual” attacks like ARP-spoofing if he is in the same local network with attacked machine.

Now, with development of cryptography science, network traffic is no more a security issue (if standard algorithms have been correctly implemented). And modern vulnerabilities, that lead to sensitive data leaks, are not related to network traffic decryption, but usually use vulnerabilities of software used to generate it or vulnerabilities of environment, where this software is executed – e.g. operating system or hardware.

For example, one can gain an ability to remotely execute arbitrary code on attacked platform and enhance its privileges. This may lead, for example, to messages capturing directly from keyboard or encryption keys can be stolen directly from memory.

Since instant messaging (IM) is now the significant part of both private and business communications, it is necessary to have an option to enhance security level not in terms of a protocol, but in terms of overall security.

Solution proposed

Since the problem is complex, the solution will also consist of several parts working together. First, we know that there are two general points of attack – software itself and its environment. These problems are solved separately.

When we talk about software environment, we usually mean hardware platform and operating system. It is clear, that protection of closed-source operating system is an unsolvable problem, or, being precise, a problem without a full solution. There were a number of attempts to create antiviruses or firewalls, but there were numerous examples of these security methods to be inefficient.

So, speaking about a secured IM solution, we surely mean custom hardware and custom operating system. To lower the possible number of vulnerabilities, we state that this operating system executes only a single process in user-space (IM software process). This will guarantee, that no other software vulnerabilities will affect the security of IM.

Second, we, of course, have to deal with a number of system-level services. It is economically inefficient to develop software without any environment. There always will be reusable pieces of software, e.g. DNS clients, TCP stacks, etc. It is also economically inefficient to protect each of them separately. But the only thing we want to be sure is the following:

- Instructions from these processes, regardless of level of execution (privileges), can't read or write memory of the IM process;
- Instructions from these processes, regardless of level of execution (privileges), can't read or write memory of kernel operating system process;

Any violation of these rules should cause a hardware exception. In this case we will be sure, that, even if an operating system subservice is vulnerable and a successful attack was performed, there will be no option to access IM process memory, where unencrypted messages and encryption keys can be stored.

It should be noticed, that IM process itself can access memory of system services for both reading and writing because it needs to pass the data and read the data (send and receive network packets, for example).

Using this concept, we can re-use traditional implementations of system services.

The third point is addressing hardware vulnerabilities. There were a number of cyberattacks, that successfully modified boot sector of persistent storage and, when hardware is started, a malware code has been executed before the operating system. This may lead to catastrophic results, because this malware code can be executed in parallel with the real operating system and is not controlled by its restrictions.

The only option it can be addressed is bootloader control by the code, that is stored in one-time-programmable memory. This kind of memory can be modified just one time physically. It should contain a bootloader copy or a hash of a bootloader, and each time the hardware is started, this code is executed before a bootloader to check its integrity.

The overall environment components and restrictions are shown on fig. 1.

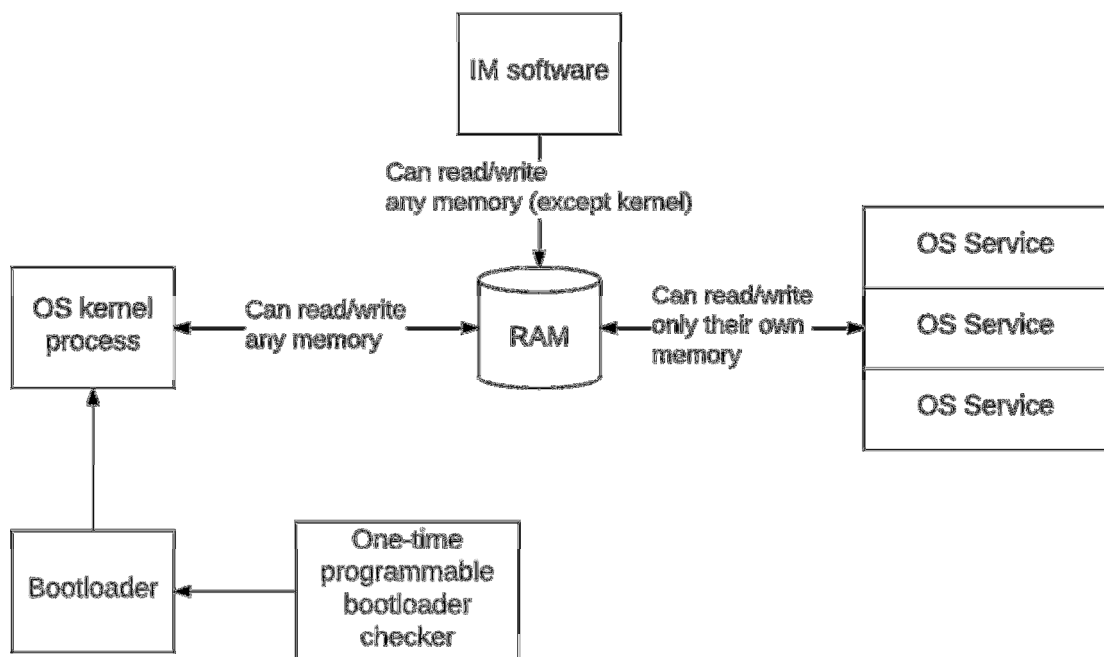


Fig. 1. Secured environment for IM software

The second point of vulnerabilities is IM software. Due to errors in software, we cannot guarantee, that it can't be attacked without special tools.

Here we should understand, that there may be two types of attacks: attacks that use logical vulnerabilities (e.g. when errors in software are "logical", the simple example is a case when protocol has a non-authenticated request "send me full database") and attacks that use stack overflow/heap corruption (e.g. errors related to lack of controls/checks over data). The first kind of errors is not a subject of presented invention. Let's address the second type of errors – stack overflows/heap corruptions. Stack overflows (and heap corruptions) are situations, when data packets go beyond its supposed limits. For example, programmer may assume buffer to be 100 bytes long, but attacker sends more and the additional data, if no checks were performed, can damage other buffers or even execution instructions in RAM.

These situations, if occur occasionally, usually lead to undefined behavior or crashes. But if used by attacker, malformed data can be structured so to predictably damage other variables or instructions and execute arbitrary code or disclose sensitive data.

To address this issue let's divide all the data into three categories:

- Fixed-length data (like numbers) – these data are safe, because modern compilers warn developers if they try to write longer value in to a shorter buffer;

- Variable-length data, but the origin of data is internal – such data buffers may lead to memory corruption, but attacker cannot influence the content of such data;
- Variable-length data, originated externally (e.g. incoming message) – such data buffers may lead to memory corruption and content of data can be controlled by attacker.

Obviously, we should protect the third type of data (and protect it system-wide, not relying on programmer's skills and attention). The solution for this is proposed:

- For each buffer of third type, we allocate 1 memory page before the buffer and one memory page after the buffer. If these pages have been accessed (that can be noticed by OS), application should instantly terminate.

Following this instruction, we will be sure, that if any overflow happen in such a buffer, we would terminate execution before the situation becomes dangerous. It should be noted, that if some other buffer will overflow and corrupt the third-type buffer, a termination will also occur, so we can assume buffers for sending data also to be of third type.

Last, but not least, issue is so-called “side data leaks”.

Side data leaks mean that without decrypting network traffic or hacking a device or application, attacker may gain information about the communication.

A simple example – if attacker sees on the network (being in the same local network attacked device is or wiretapping transmission lines) outgoing packets from the device, then he may assume some communication is happening. Also, analyzing the traffic dumps attacker may understand the type of communication (sending files, messaging or voice calls), intensity of communication in case of messaging, duration of session in case of voice calls, approximate size in case of files.

All these examples do not directly lead to sensitive data being leaked, but can be very dangerous in context. For example, if police officer starts intensive communication, it can mean that he is planning operations against criminals, and if his traffic is wiretapped, even not knowing the actual content of messages, criminals may guess that they should relocate and stop their operations or look for insider.

Previously, these issues have not been taken into account when secured IM software was developed. But if we have a robust and secure environment and application, side data leaks may become the most vulnerable part of the system and side-leaks analysis – economically efficient way of attacking application users. The situation becomes even more dangerous, if such application is used only for sensitive talks – it means that there will be no “noise traffic”, like talks about weather or holiday plans. All the traffic on such application becomes a thing of interest to attacker and every message means something.

The proposed solution for this issue is a protocol obfuscation approach. Protocol obfuscation is a “best try” to make the traffic both untraceable (hard-to-extract from other network packets) and unanalyzable (hard to distinguish packet types, even knowing exactly the algorithms of software used to originate them).

The proposed instructions for protocol obfuscation are based on the fact, that we actually have three types of traffic:

- Voice calls – small fixed-sized packets sent frequently;
- Messages – medium packets sent rarely (“rarely” here means that these packets are usually sent with the speed of human typing multiplied by the average number of symbols in messages);
- Files – large packets sent very frequently, one-by-one;

It is quite obvious, that we can’t make these three types to look same (or it will require an enormous amount of parasite traffic that we should generate). But it is an option to, at least, mask the traffic with fake packets sequences of each type.

Let’s address the intractability.

Protocols tracing is performed with Deep Packets Inspection software. It analyses a bunch of characteristics – addresses of source and destination, sizes, frequency, etc.

There may be two things we can perform to make our protocol untraceable:

- 1) Use VPN for every transaction. It will lower the speed, but DPI won’t be able to trace destination addresses, because everything it will see is encrypted traffic to VPN server. The alternative to this is port jumping and protocol jumping, that significantly reduces the quality of service.
- 2) Use parasitic packets payload to mask the actual packet size.

The first principle is fairly simple, let’s address the implementation of the second one. Let’s assume we have 3 types of packets – 3, 4 and 5 bytes long, and attacker knows this also.

To mask these packets, we should add 2 bytes to the first packet type, 1 byte to the second type packet and 0 bytes to the third packet type.

This approach sounds simple, but in real system may result in enormous amount of parasite traffic generated. For example, average VoIP packet is approximately 100 bytes long, while the largest packet can be up to 3000 bytes. Adding 2900 bytes to VoIP packets is unacceptable.

This leads us to the mixed solution – let’s not only increase packets, but also split too long packets into smaller parts. For example, let’s divide 3000-bytes long packets into 10 parts 300 bytes each and add 200 bytes to VoIP packets. This will significantly decrease

the amount of parasite traffic in VoIP packets (more than 10 times) and just slightly increase the overload on large packets.

The second task is to make our traffic unanalyzable.

We should, assuming the attacker knows the exact algorithm, generate fake packets and send them over VPN. Moreover, we should add a fake responder, so that attacker would see the responses on our traffic.

How often should we send fake packets of each type? Let's assume user sends 1-5 real messages per day. We should send at least the average of packets per day of fake packets during user activity time and at least 2*average of packet per day during inactivity to hide the real user activity time.

The same situation with files and calls. With calls we should also use fake durations which should be average of user's duration +/- standard deviation with normal distribution probabilities.