Secure Integration of Multiprotocol Instant Messenger

Sebastian Bala
Institute of Mathematics and Informatics
University of Opole
Opole, Poland
Email: sbala@math.uni.opole.pl

Tomasz Wasilczyk
Google
Mountain View
Email: tomasz@wasilczyk.pl

***Abstract*—Pidgin communicator is a multiprotocol instant messenger client, developed by open source community. It was originally a third party client of AIM (AOL Instant Messenger) protocol for Linux operating system founded in 1998. At present, its main goal is to provide common interface for every protocol it supports, so the user does not need care about protocol to use or peer's IM identifier, while he just want to talk with a Bob.**

The work focuses mainly on security and functionality aspects like password storage strategies, integration with plugins which provide privacy of communication. This paper shares experiences has been gained during the process of developing and replacement of existing code for version of Pidgin 3.0.0. The project realizes the list of suggestions, mainly concerning security, which has been created after code review and provide some new technical solutions that can be implemented in the future versions.

***Keywords*—instant messaging; off-the-record; password storage; libpurple library; software integration**

I. Introduction

Instant messenger is a program making real time text-based conversations possible over Internet. As the name stands, messages entered by the user are almost instantly delivered and shown to the recipient. In most cases, the text is sent using service provider's servers, which keeps the connection with both sides of conversation.

Many service providers use its own, proprietary protocols to communicate between client's instant messenger application and the server. The set of computers accessible with that server (or servers pool) is called *IM network*. There are cases, when two distinct IM networks use the same protocol (for example, ICQ and AIM networks both use Oscar protocol), but not the opposite.

Every user that wants to communicate using selected IM network needs to register his own IM account. To write a message to another user, you have to know his account identifier, for example identifiers: 12341234 (ICQ), bob123 (Yahoo!), bob@example.com (XMPP), which may be numeric or alphanumeric, depending on IM network.

The XMPP is an example of open IM network, where many service providers takes part. In particular, every host may join the network with little effort – all he need is to set up the server and put some DNS records into his domain configuration. XMPP identifier (e. g. `bob@example.com`) is composed of service provider host name (`example.com`) and the user name (`bob`), unique within the scope of a domain. That makes it possible to communicate between users using different service providers. Some email providers (like GMail) offers also an XMPP account, making both addresses the same.

Multiprotocol instant messaging applications gives ability to talk with users of various IM networks, without installing a client for every single network. However, the user still have to register an account in every network he wants to use. Many multiprotocol instant messengers supports proprietary IM networks by reverse engineering its protocols, which design is kept in secret by its founders. Therefore, third party clients may not work exactly like the official ones – that may lead to minor problems, especially when using less popular protocol features.

Pidgin [1] (formerly GAIM) is a multiprotocol instant messenger client, developed by open source community. Its main goal is to provide common interface for every protocol it supports, so the user doesn't need to care about protocol to use or peer's IM identifier, while he just wants to talk with *Bob*.

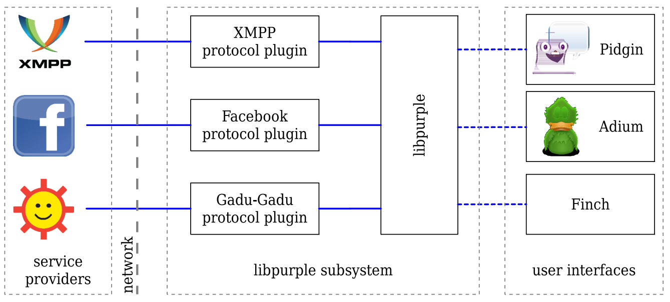

Fig. 1. Schema of Pidgin project. Only one user interface at a time is possible.

The Pidgin project consists of the *libpurple* library, which provides protocol support, and currently, two user interfaces:

This publication is a part of the project funded by the National Science Center based on the decision no. DEC-2013/09/B/HS5/00019

[1]In this paper we use some logos and names of which marks existing software. The only reason of using it is explanation how mentioned software use libpurple library, how libpurple intercommunicate with mentioned software or which software is written to which operating systems. Our actions are outside the commodity trade, and we do not profit from advertising any products.

Pidgin (GTK+ based, for Linux and Windows) and *Finch* (for Linux console). There are also many third party instant messengers based on libpurple: Adium (Mac OSX), Instantbird (using Mozilla's Gecko engine) and few others. That means, all libpurple library improvements or security drawbacks affects all these applications, so it's even more important to take a good care of it. The principal idea of libpurple core is to provide thin interface between protocol plugins and UI. Plugins handle majority of topics: protocol support, TLS connectivity, ciphers implementation, message logging, user interface (actually, UI is not exactly a plugin – it connects to libpurple as a back-end).

II. Project

The main part of the project is involved with security improvement of libpurple library. Security and network part of the work can listed as follows:

Finish and merge keyring feature branch which fixes one of the most complained security issues in Pidgin – storing plaintext passwords for IM accounts in settings files.

Provide new HTTP implementation to replace old, error-prone code.

Make new End-to-end encryption API, providing common way to present or alter the security level of conversation to the user.

Integrate Off-the-record messaging plugin with Pidgin main release. That makes secure instant messaging easily reachable for all its users.

Progress of the work has been documented on the blog [1].

III. Password Storage and Keyrings Usage in Pidgin Communicator

When user configures his instant messenger client, he needs to set up every account he want to use. This includes choosing the IM network, providing login for the existing account and optionally password. After every startup, IM client signs into all previously configured accounts. This includes connecting with server, passing the login and authenticating it with password. In most cases, authentication phase requires password in its original (plaintext) form. If the user doesn't provide the password while setting up an account, he will be doing it for every account and every time the program starts up.

The above may be annoying for the setup with many configured accounts, so most of users decide to store passwords for all their account. However, this may be a security issue, because libpurple 2 saves them in plaintext form in *configuration files*. If someone has access to user's directory, he will easily obtain all stored passwords.

Current Pidgin security policy is not to store passwords by default and optionally store them in plain text. The better solution would be to *safely* store these passwords, but doing it was nearly not possible, due to libpurple's API constraints.

For instance, libpurple provides a method to get a (plaintext) password for certain account. It's synchronous, so the result have to be returned at the end of a single function call. On the other hand, some of the password storage methods doesn't provide a synchronous API (e.g. when it needs to ask the user for a *key*). There is a possibility to implement it in synchronous way: waiting in a `purple_account_get_password()` (the libpurple function to get password for certain account) function until the asynchronous request is finished. Unfortunately, this would lead to a lock in a single-threaded application.

To solve the problem properly, the `purple_account_get_password()` function have to be replaced with the asynchronous version. However, when a library uses SemVer [2], every removal or a change in any publicly exposed function causes a major version bump. So far, the whole Pidgin project did only one such increment (as of 2013) since its first *stable* version in 2004. Ultimately, it wasn't possible to change this behavior in Pidgin 2.x.y branch with respect for the SemVer rules.

Apart from safe storage, there are also other problems directly related to account passwords. In case of careless implementation, there is higher chance for its disclosure, either with intentional action from the third party or even without it. One of the aspects is placing plaintext passwords in debug logs. Protecting it is not as simple as making a statement, that no passwords must be printed using `purple_debug_info()` (the libpurple function that adds a line to debug log). Let's take a case, when an HTTPS request is done to perform an account sign in. For the sake of debugging HTTP requests, its contents is printed to the debug log. As a result, the password is revealed in the log, despite treating it carefully by the protocol plugin, because HTTP subsystem knows nothing about confidentiality of its requests. Thus, `purple_debug_is_unsafe()` is used to determine, if any text that *may* contain sensitive data, should be printed to the log. Then, in the runtime, the user may switch printing such data just by setting `PURPLE_UNSAFE_DEBUG` environment variable.

Another serious issue is the possibility for operating system to move a memory block containing a password to a swap file. Even, if an access to the file is restricted by the system during its operation, someone with physical access to the machine can connect the hard disk drive to another computer and read the file directly. Password retrieval from the swap file is not trivial, because its contents contains raw dump of memory blocks, but it is still possible.

GnuTLS is one of the libraries that takes care of this, by marking a memory range, that must not be moved to the swap file. Every variable that contains private key or the information that may reveal it, is placed only within such memory block. Thus, the problem is solved. Libpurple could implement similar solution, but it requires deep code refactoring and a lot of effort.

Every bigger software project has its defects. There is virtually no possibility to ensure, that a program is free of bugs, but we can minimize the risk of harm, when it occurs. Specifically, an attacker may cause some kind of remote, partial memory dump, which may contain plaintext passwords. If an application holds them for a long time and in many copies, the attack will be easier. A simple rule, to hold passwords in memory no longer than needed, could make some percent of attacks unsuccessful.

There are also cases, when an instant messenger client cannot do much with passwords security. For example, some IM protocols sends plaintext passwords during authentication process. The only thing we can do is to enforce SSL connection, if the protocol supports it.

Various instant messaging clients treats differently user's private data confidentiality. Some of them do a good job properly using solid encryption methods, but there are also bad ones, which just pretend that they protect the data.

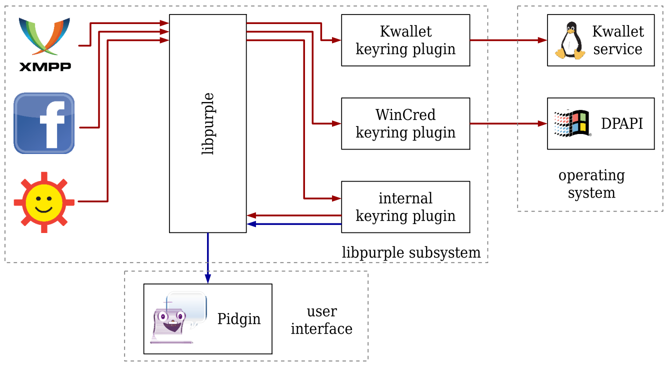

Fig. 2. A diagram describing password request flow (red path): initiated by a protocol plugin, libpurple routes it to the currently configured keyring plugin, which may forward to the external password storage service or back to libpurple (in secured form). Keyring plugins may interact with user through libpurple API (blue path).

Keyrings are credentials management tools, which provide centralized solutions for password storage. There are many various alternatives, but all of them have one thing in common: IM client (or any other software that deals with user credentials) delegates password storage to the keyring, so it doesn't have to deal with the security of stored data.

Keyrings in Pidgin are designed as plugins, implementing at least two methods: *storing* and *retrieving* a password for a certain IM account. Credentials may be encrypted and sent back to Pidgin configuration files, or stored entirely externally. Thus, such plugin may, or may not implement *export* and *import* callbacks, used for associating data needed for password retrieval with the account. It may be anything: AES-encrypted passphrase, unique identifier of a record in external database, or even nothing. For the old Pidgin versions (or an internal keyring with encryption disabled), the *extra data* is just a plaintext password.

Example III.1. A structure of account-related data, saved in preferences file:

```
<account>
<protocol>prpl-example</protocol>
<name>alice</name>
<password keyring_id="internal"
mode="cleartext">password1</password>
</account>
```

Typical keyring lifecycle is as follows. When libpurple is starting, all plugins (including keyring ones) are probed and loaded. The plugin checks, if all the necessary dependencies are satisfied (i.e. if the keyring database is installed in the system) and it eventually registers a new `PurpleKeyring`. Then, libpurple selects the configured keyring (user may choose, which one to use) and *imports* all previously stored extra data for each IM account, to the keyring's internal memory. When libpurple wants to authenticate, it needs its password, so it *reads* it from the keyring. This process is asynchronous, because the keyring may not be able to respond immediately (there is no possibility to freeze current thread, because Pidgin is single-threaded, with some exceptions) – i.e. when it needs to ask user for the master password, or make an asynchronous system call.

Example III.2. A lifecycle for the internal keyring (without encryption enabled) is as follows. At startup, it *imports* a plaintext (for any other keyring, it would be a ciphertext) password from the preferences file and stores it in keyring's internal memory (in this case, it's a simple hash table). When the password is necessary, it just *reads* the password from its cache.

The opposite way is pretty symmetric. When user updates password for a certain account, libpurple *saves* it to the keyring. Like in the *read* case, its asynchronous, but caller doesn't need to wait for the result – it's keyring's responsibility to correctly arrange all requests. Afterwards, libpurple schedules saving account preferences to the disk. When processing every account, password-related data is *exported* from the keyring and saved in the preferences file.

A. Implementation Difficulties

There were attempts to provide keyring support for legacy Pidgin, but they had significant drawbacks. The most popular plugins that implemented such a feature, cleared a password before saving it to configuration file and restored it (from external database) after loading the account info.

The main issue was the libpurple's approach to passwords availability. The library (and all its protocol plugins) assumed, the password is directly accessible, without any need for waiting. On the other hand, most of keyrings returns decrypted passwords asynchronously, sometimes after a long delay (e.g. when it needs to ask user for permission). This issue together with a fact, that the clearing/restoring method was a bit hacky, were prerequisites for a complete redesign of password-related code in libpurple. It required API changes, such as making password access methods asynchronous. Thus, it wasn't possible within libpurple 2.x.y branch.

The most interesting part has been done is designing internal encryption schema, which is described in the succeeding paragraphs.

Most keyring plugins (for example KWallet, GNOME Keyring Secret Service, DPAPI) depends on external libraries or operating system capabilities. For the time of writing this document, there were no keyring systems, that could work on every platform supported by libpurple.

We needed to provide a solid, cross-platform plugin that could be used as a default keyring. This one implements the whole schema internally, without any additional dependencies. There are two modes of operation: encrypted and unencrypted. The first one uses some decent, standard algorithms, such as AES or PBKDF2. Its details are described in the following paragraphs. The latter one works almost the same as in old libpurple – storing plaintext without any modification. Thus, it provides some level of backward compatibility.

Internal keyring has some certain pros and cons, compared to the other keyrings. It's perfectly portable, as all the passwords are kept within libpurple's configuration files. On the other hand, it makes no distinction between sensitive data (passwords, even encrypted) and other settings, so the user could share it unintentionally. There is another drawback of such solution: the keyring is an internal part of the client, so there is no system service. That means, the master password must be entered once per every application launch (instead of once per system boot) and cannot be integrated with system logon.

B. Implemented Encryption Schema

In secured mode of internal keyring, passwords are no longer stored in its plaintext form. Instead a base64 representation of its ciphertext form is stored in the same node of the configuration file. The detailed schema of a ciphertext is as follows:

$$C = IV \parallel \text{AES}\left(P \parallel p_1 \parallel c \parallel p_2\right)$$

where C is the ciphertext, IV is an initialization vector, P is the plaintext password, p_1 is a minimum length padding, c is a control string and p_2 is a pkcs7 padding. The IV is a random, 128bit vector, which makes every ciphertext different, even for the same plaintexts and keys. To make things a bit simpler, it is stored together with the ciphertext, as its prefix.

The cipher used in this implementation is AES-256. Its blocks are 128-bit long and symmetric keys are 256-bit long. AES is a strong cipher with many papers proving its security, so we won't analyze it here. Instead, we will mention a few pitfalls that could be committed using it. We use the CBC encryption mode, which is currently 'a standard industrial practice' for the password storage [3].

Another hypothetical issue is choosing a bad pseudo-random function for encryption key or IV generation. Some of them could be predictable, which is bad for encryption keys. Others could result in a heavily reduced space of possible outputs, which can be catastrophic for IV. There are good sources for entropy, like ambient temperature or mouse input. On Unix systems, they are easily accessible via `/dev/urandom` device. For multi-platform applications, libgcrypt provides `gcry_random_bytes_secure` routine for the same effect.

The c control string is a simple way to verify, if decryption was successful. It might decrease the security level, by allowing some kind of known-plaintext attacks. However, it doesn't, thanks to a well-randomized IV and the fact, it's at the end of encrypted block.

The last part is a PKCS#7 padding. It's necessary, because AES is a block cipher and the encrypted data length needs to be a multiplicity of a block length.

IV. Off-the-Record Messaging

Off-the-record protocol [4], [5] consists of several components. The most important are about message encryption and key exchange, but the smaller ones are crucial for hassle-free usage. OTR doesn't aim to provide a full-stack messaging solution. Instead, it gives an overlay for any underlying messaging protocol, which adds privacy. Obviously, not every single messaging protocol could support OTR supplement for its service, so it needs to implement its features with almost no assumptions on a format of messages that could be transferred. After all, the protocol could run on top of primitive protocols like IRC or even SMS.

OTR, just like many other secure messaging protocols, provides *encryption* with a well known AES128 algorithm in CTR mode. AES is already well-described blackbox, so the only thing to care about is a good key for symmetric encryption. Socialist Millionaire protocol (SMP) *authenticates* [4] one or both parties. Some clients doesn't implement this feature, so it's not possible to verify other party identity, but it doesn't affect other features.

The authentication is totally repudiable – there are no digital signatures, only MACs. The user can *deny* [4] he sent any of the messages – there is no mathematical proof of that.

Private keys are not directly used for message encryption. Instead, a short-living encryption keys are distributed before every conversation and securely erased later. Thus, even if a long-term keys gets compromised, an attacker can not access previous messages – that's what it is called *perfect forward secrecy*.

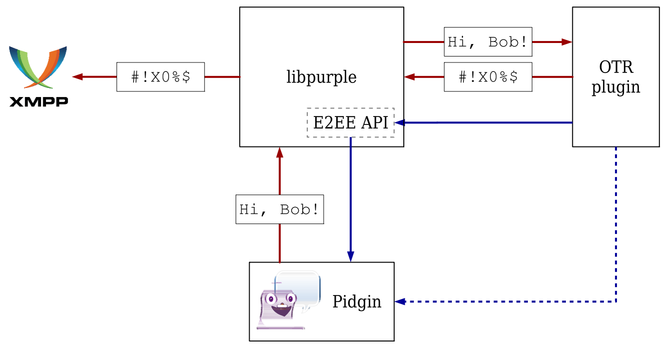

Fig. 3. Interoperability diagram for OTR plugin. Plugin captures outgoing messages and encrypts them on-the-fly (red path). OTR interacts with user via libpurple APIs (blue path). Previous solution of directly using Pidgin APIs resulted in hard dependency on that UI.

When working on OTR, it was a plugin for Pidgin, which provided all expected features just fine. The original task was to push it to Pidgin source tree, so it could be released (and possibly enabled by default) with the official client. It could be scheduled as following: copying pidgin-otr sources to its own subdirectory in Pidgin source tree (trivial); fixing possible compilation issues, caused by differences in C compiler configuration for both projects (like different sets of enabled warnings); adjusting Pidgin buildsystem to compile pidgin-otr too – includes detection and import of its dependencies, like libotr; providing a way to auto-enable OTR plugin when necessary.

The hard part was auto-enabling, which would mean the following: by default, Pidgin doesn't support OTR, so it can't encrypt outgoing messages or interpret incoming ones. But we could check incoming messages for existence of `?OTR` tag and ask the user to enable pidgin-otr plugin. Another option would be just to enable pidgin-otr by default (user could still disable it). However, taking the easy route could waste the opportunity to convert this Pidgin plugin to libpurple plugin. The difference is subtle, but important: all libpurple plugins do work with Pidgin, but Pidgin plugins do not work with other libpurple UIs. From among the four most popular libpurple front-ends, only Pidgin has full OTR support and Adium has limited (without SMP method implemented). Libpurple OTR plugin gives any libpurple-based client full OTR support – even for console-only Finch. Let's focus on causes, why such conversion is possible at this exact moment – the *moment* (it was meant to take few months between API unfreeze and 3.0.0 release, but it takes few years till now) of libpurple 3.0.0 API breakage – and why it was a non-generic Pidgin plugin before. Writing a libpurple plugin gives an important perk of having a support for all libpurple-based instant messengers at once. On the other hand, it's greatly limited compared to Pidgin plugins (or Finch plugins), as it can't modify user interface. It doesn't mean libpurple plugins can't interact with user. It means, they need to make it through libpurple.

Taking a change of 3.0.0 API breakage, We decided to extend libpurple's API in such way, Pidgin OTR needs it. These changes are generic, so other plugins may have a benefit from that too. Some of them were subtle (but meaningful for improving user experience), others were unavoidable to bring all OTR features: new End-to-end encryption providers API: a generic interface to display security level for opened conversations (like: encrypted, unsecure etc.); Request API refactoring with new PurpleRequestCommonParameters class; further Request API improvements, like a possibility to format text with HTML, altering dialog icon, setting a *help* action, providing extra actions, better linking to the parent window; adding a new window type of *please wait* dialog, with an optional progress bar.

The hard part is, all of these changes applies to the whole stack of APIs: libpurple as the back-end and both Pidgin and Finch as front-ends. Moreover, all third party front-ends (Adium, Instantbird) needs to implement all of it too.

The next goal was to provide API rich enough to cover all OTR-related scenarios, but generic enough to fit other encryption plugins. OTR has four possible privacy levels [6]: not private (not using OTR), unverified (using OTR, but the other party's public key was not verified), private (using OTR with verified public key), finished (the previously ongoing private conversation has been terminated).

We made decision to delegate the responsibility of defining these levels to the encryption-related plugins. OTR plugin needs to register itself as a E2EE encryption provider, provide a list of used privacy levels and set a callback for creating a privacy features menu. Then, it notifies the conversation object every time its privacy level changes.

There arise another problem: what happens, if there is more than one E2EE encryption provider? Both could define disjunctive privacy level sets, which could complement each other, or not.

Example IV.1. Your IM service may be configured to use secure tunnels to communicate with its clients, or not. We can imagine a plugin, that defines two privacy levels: fully encrypted chain (that every link between you and your buddy is done by encrypted channel) and unencrypted (so there is a link, like between you and the server, that use unsecure connection).

Such set is partly disjunctive with OTR's, but we could define an order among them (from the one user might like the best, to the one user should worry about the most): private, unverified+encrypted, encrypted, unverified, not private and unencrypted (the last two are of equal priority). Finished conversation is out of order.

We could combine outputs of all registered encryption providers into one state and display the best fit to the user. However, the rules of combining security levels of different domains might be complex when we take into account two already defined, specific sets (just like in the example). It becomes incredibly complex when we have to provide a generic rule, for any level sets combination. Thus, we made decision to allow only one E2EE encryption provider registered.

It is a limitation, but only a small percent of users actively use more than one tool for encryption. We will reconsider this decision when somebody create a third party plugin that conflicts with OTR.

V. NAMECOIN

There was a suggestion [7] to implement another kind of public key verification mechanism. It was meant to be based on Namecoin, a fork of Bitcoin. Both are cryptocurrencies, but we will focus on their opportunity to store data in a distributed network.

Two already implemented OTR key verification methods covers many use cases, but they might not be convenient enough for some. Manual verification requires a secure channel to send a SHA fingerprint[8] to the other party. SMP method requires having an unique secret with any peer we would like to talk to. Both are doable, but the effort might discourage people from using it.

Let's consider the following scenario. You are an instructor, who has a lecture with a large audience. You want to have a private contact with anyone who would like to reach you after the speech (for example – for students who need to submit their exercises). Assumptions are the following: audience (students) trusts no third party that could distribute your public key; they have no proof of your identity, other than you as yourself standing at rostrum; you don't need to verify identity of your students (they have no interest in fraudulently submitting their work not as themselves), but they need to verify they are speaking with you (to not submit their work to other, dishonest student).

Using manual verification method, you would need to write your 40-characters long SHA fingerprint – using a chalk on a blackboard – which is awkward and error-prone. It is still better than SMP method, where you would have to establish and exchange an unique secret with every single student. It would take unreasonable amount of time. An assumption of no trusted third party makes it impossible to use a CA signed certificate, which could be handy in this case. On the other hand – quite expensive.

Namecoin provides a semi-persistent key-value storage. Any network peer can register any unused name (key), paying a tiny fee (0.01 NMC, which is around $0.004, as of 2015). It expires after roughly 200 days, unless user updates or renews it. After initial registration, only the name owner can make changes to the value stored at the specific name.

Just like in the Bitcoin case, all the data is stored in the blockchain, shared by network nodes. However, Namecoin allows storing any user-specified, transaction unrelated data. The only limitation is maximum data size – it may not exceed 520 byte limit. It's enough to store SHA fingerprint, or even the whole public key. Hypothetically, larger public keys could be split and stored at multiple names, but it is enough to send the public key over an unsecure channel and verify it with a fingerprint.

Let's go back to the professor-student scenario we just considered. You could register any available human-readable name, like `johndoe`. Technically, OTR plugin could have its own namespace, to avoid conflicts with other uses. Thus, so your Namecoin key would be `otr/johndoe` and contain your public key SHA fingerprint.

Then, you could *broadcast* it, by just writing your `johndoe` identifier on the blackboard. Every student could verify your public key with the Namecoin stored fingerprint. It is not a secret, so there is no way for any student to trick any other spoofing your identity (at least, basing on a knowledge of the name).

Ultimately, we have not implemented additional Namecoin-based OTR key verification method. It might be a good idea, but it doesn't have strong perspectives of being widely used. Instead, user can manually send the OTR key fingerprint to the Namecoin network and retrieve it with moderately small effort, using existing tools. Our implementation could just make this process faster and easier. Thus, we decided to spend time on other security-related improvements.

However, we have extended OTR plugin with *Fingerprint verification API*, to make it possible for any developer to implement it as a plugin. It was quite tricky, because OTR is already *a plugin*, so we would want to provide *a plugin for a plugin*.

Pidgin and its plugins are written in C, so they need to not only be API compatible, but ABI compatible too. It means that Namecoin plugin compiled with specific OTR plugin version would be compatible only with that version (or set of versions). Moreover, libpurple would have to track dependencies between plugins, which is not implemented yet – otherwise, ABI incompatibility could lead to crash of OTR plugin.

We had to come up with a solution that satisfies the following assumptions: make no strict requirements about OTR plugin version, so any *verificator* compatible with Fingerprint verification API would work with any OTR plugin version compatible with that API too (it still needs to be ABI compatible with libpurple itself); gracefully fail in case no compatible OTR plugin was found; API should be expandable without compatibility breaks; easy to use (eg. to implement a plugin).

VI. Conclusion

After doing some work on security-related topics, it is worth to assess its condition before and after taking changes. We can not say Pidgin project was insecure before, as any security bugs are being hotfixed and released in 2.x.y branch. However, we could take a look at wider definition of security. All the improvements has been done were not a revelations, but all together could bring Pidgin project to a next level of security applications. In fact, we needed them to catch up the top notch secure instant messengers. While user could live with the patched and working client and say it is secure, then new features might extend sense of security in other dimensions. Keyrings implementation allowed to weaken constraints for environment considered secure (so user could put his passwords on unprotected storage). Off-the-record integration resulted in strengthened implications of private messaging.

References

[1] https://blog.wasilczyk.pl/en/tag/pidgin-security-grant/, May,2017.

[2] "Semantic versioning," http://semver.org, July, 2015.

[3] P. Gasti and K. B. Rasmussen, "On the security of password manager database formats," in *Computer Security - ESORICS 2012 - 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings*, ser. Lecture Notes in Computer Science, S. Foresti, M. Yung, and F. Martinelli, Eds., vol. 7459. Springer, 2012, pp. 770–787.

[4] N. Borisov, I. Goldberg, and E. A. Brewer, "Off-the-record communication, or, why not to use PGP," in *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society, WPES 2004, Washington, DC, USA, October 28, 2004*, V. Atluri, P. F. Syverson, and S. D. C. di Vimercati, Eds. ACM, 2004, pp. 77–84.

[5] https://otr.cypherpunks.ca/Protocol-v3-4.0.0.html, July, 2015.

[6] "Otr levels," https://otr.cypherpunks.ca/help/4.0.0/levels.php, July, 2015.

[7] https://pidgin.im/pipermail/devel/2013-August/023151.html.

[8] "Fingerprint," https://otr.cypherpunks.ca/help/4.0.0/fingerprint.php, July, 2015.