Secure authentication through standard storage devices via Pluggable Authentication Modules

Erik Sonnleitner
0555464/911
JKU Linz

Contents

Abstract

Multi-factor authentication doesn't constitute a revolutionary concept nowadays, although its occurence on ordinary desktop systems is quite limited. Nevertheless, authentication through ownership-based methodologies currently need special hardware in most cases, e. g. an external smartcard-reader. To advance one step beyond this limitation, this paper describes a software suite and the corresponding techniques to use commodity storage devices and especially USB mass storges drives for achieving cryptographically strong authentication.

1 Introduction

1.1 Motivation

At least since global interconnection of computational resources has been made accessible and affordable for non-military and non-corporate customers through the origin of the Internet, the term of *security* has been conventionalized to a buzzword in information technology. Beyond the great potentiality of digital intercontinental real-time communication, global addressing schemes and clever routing algorithms don't come unheedingly: Whether aiming on malicious code (re-)distribution or clandstine network attacks, the existance of a global network provides enough sources of danger for introducing a security hype.

The *secure socket layer* (SSL, nowadays more commonly known as *Transport layer security*, TLS) allows us to establish cryptographically secure connections to our trusted site of choice, in a completely transparent manner and without the need for explicit user interaction (the latter is, in fact, the reason for the revolutionary breakthrough of transporting sensitive data across the Internet). Cryptographic appliances on the transport layer of the OSI model (and their common implementations like TLS) have been heavily discussed for a while, but the tendency of using those techniques still grows [Hil07].

But even when virtually every node on the Internet may gain connectivity to a victim of choice, the majority of attacks comes from within an Intranet, according to [Sch04]. Such an interior network will mostly be a LAN, either wireless or wired. The possibility of directly accessing a network on layer two of the OSI model provides a much broader attack surface; especially techniques which rely on sniffing as well as

several packet injection methods like ARP cache poisoning may only get successfully executed at network level. Also attacks which force to abuse higher-layer concepts like TCP session hijacking are quite hard to accomplish without physical network access (if the attacker doesn't actually own a router on the packet's way to the victim and vice versa).

Beyond the realm of networking and associated security concepts, authentication itself as a designated identification and verification process, still remains as an incredibly important application-area on unconnected workstations and servers.

Nevertheless, with or without the application of strong cryptography, the actual methods of authentication havn't changed very much. At a generalized scope, three primary approaches are currently predominant:

- Knowledge. Something the user who wants to gain authentication needs to provide and nobody else knows. This is by far the most widely used method, including authentication mechanisms based on passwords, passphrases, PINs, etc.
- Ownership. The user who wants to gain authentication must provide a physical object nobody else possesses. This is the main idea behind smartcards, RSA tokens and (partitially, at least) even RFID transponder chips; but also ordinary doorkeys may be seen as such a token (without an underlying software mechanism).
- Biometrics. An unique biological attribute of the user who wants to gain authentication. Biometric systems are heavily discussed, since it's the only way of verifying the *identity* itself and not just an *entity* proving knowledge or providing ownership; while a password may be typed from any person and even remote computers, biometric attributes are usually harder to replicate.

Authentication systems which are conceptionally designed to provide very strong authentication are frequently capable of using *two-factor-* or even *multi-factor-authentication*. These systems are e. g. common in high-security environments like in beaconage. The execution flow of a cash dispenser for example, needs to include permissions to read out the bank card (which represents the ownership part), as well as the submission of a personal identification number (which represents the knowledge part) to provide idendity verification and performing transactions.

Early forms of authentication through proving knowledge were mostly limited to a password. While the principle itself remains the same, several internal structures have changed. Global internetworking built the foundations of secure replacements for classical tools like `rlogin`, `rsh` and `rcp`, since unencrypted password transfers are no more sustainable in global networks. But also without consideration of conceivable implications of network sniffing, the server-side password management has improved. As early Unix implementations stored cleartext passwords in `/etc/passwd`, nowadays Unix authentication mechanisms amost exclusively only store cryptographic fingerprints of those passwords, so that a theft of password information wouldn't reveal the cleartext passwords themselves [PC04] [Bau05]. Finally after hashing was introduced, the idea of salting became popular, and so on. See the hashing introduction section for a deeper inspection on this topic.

Passwords have the advantage of not being tangible as physical medium; nevertheless, password strings may get logged, spied out, or captured quite easily. Bruteforce attacks are widely used and mostly executed as fully automated processes. In addition, passwords may perhaps easily be guessed when collecting information about the victim. Last but not least, laziness often results in multiply used, too short and too easily guessed passwords.

The concept of ownership goes into another direction: By supplying a token to the authenticator which only the (authorized) entity possesses. Unlike passwords, therefore the authentication processes relies on a physical medium which only exists once. However theft, loss and also damage may happen to the token and should be covered under special consideration.

1.1.1 Hardware tokens

Even if commonly classified as very secure, authentication tokens suffer from another problem, namely replication. A well-designed token must not be (easily) reproducible. Normally this is achieved by manufactoring proprietary hardware tokens like those available from the RSA security division of the EMC corporation. These tokens can not be read out by standard hardware and do not have standardized interfaces, which makes the whole system harder to attack. Although standardized in size and behaviour, creditcards follow the same principle. Very-high security tokens actually have CPU and memory already built in to restrict the access to internal memory areas from *any* hardware device [PW06],[Sch04].

Using non-standardized interfaces on proprietary hardware may be seen as security enhancement. On the other hand, in spite of being dependent on only one company, authentication deeply relies on the authenticator device, like a smartcard reader. However, this may result in a nasty situation when trying to use onwership-based authentication on mobile devices, which may provide a dozen of standardized interfaces in case of laptops, but no built-in authenticator for that's the nature of proprietary hardware.

This paper introduces an authentication software suite, which is conceptionally designed to provide one- and two-factor authentication through ordinary storage devices, while maintaining every-day-use suitability and supports commonly used standard storage hardware like USB memory sticks or *Secure Digital* cards (SD).

I am not suggesting this project to be used in highly sensitive environments, due to some restristions conditioned by the nature of standard-interfaced storage devices without built-in (semi-)programmable logic. Smartcard authentication remains the more secure way; but nevertheless I'm confident the level of security reached by this project is so called *good-enough* security, according to the Gartner IT security summit 2006 [Gar06]. It may seriously enhance authentication security on Linux boxes (and possibly, other PAM-capable operating systems), while being easily usable, non-disruptive and brain-gentle since only one password has to be remembered[1][2] for all services working with PAM.

1.1.2 Advantages and disadvantages of ordinary storage media

Using normal storage media as source of authentication fingerprints is somehow double-edged. Ordinary storage media providing standardized interfacing means that virtually any write-enabled storage device may be used as hardware token to hold authentication information. On the other hand, this fact goes along with a couple of disadvantages:

- As a matter of fact we need write-support to get the authentication information onto the device. This means, that everyone who owns the device may change everything s/he wants.
- Since the device must be publicly readable, everyone who owns the device

[1] At least; it's up to you how many tokens you create.

[2] To be honest, it's easily possible to use this authentication system without any password at all; but in the end, I would not really recommend this.

may obtain a forensic copy of everything what's on the device. Therefore, the information which is actually stored on the medium must not reveal *any* useful information, hints, or data which may be used to break the authentication procedure or backtrace the initialization vectors.

Unlike smartcards, the whole memory is directly accessable and readable from every computer which has the interface' counterpart. The interesting part is to develop techniques that do not rely on the fact of data hiding and built-in hardware logic.

1.2 A quick introduction to PAM

The *Pluggable Authentication Modules* architecture or in short PAM, has been introduced by Sun Microsystems as Request For Comments (RFC) paper in 1995 and acts as an authentication middleware between

1. the *application* which forces to authenticate a user (e.g. login, secure shell, su, etc.), and the corresponding
2. *authenticator*, which can execute virtually any procedure of authentication.

The latter is represented through the actual *modules*. PAM brings quite a couple of modules out of the box, for authentication via e.g. normal Unix passwd/shadow retrieval, LDAP service connectivity or verification by special hardware like smartcard readers as shown in figure 1.

The conceptional design of PAM shows parallels to the IEEE 802.1x standard when mapping the supplicant, the authenticator and the authentication server to the calling application, the PAM framework and the authentication method (the PAM module) respectively.

The host's administrator may define granular configurations to decide which service may gain authentication through which module(s) under which certain conditions (and much more), normally stored in `/etc/pam.d/`. On Desktop systems, the default authentication routines almost exclusively rely on `pam_unix.so`, the module which implements standard Unix identity verification by hashing a given password and comparing the resulting string to the value defined in `/etc/shadow` for the desired user.

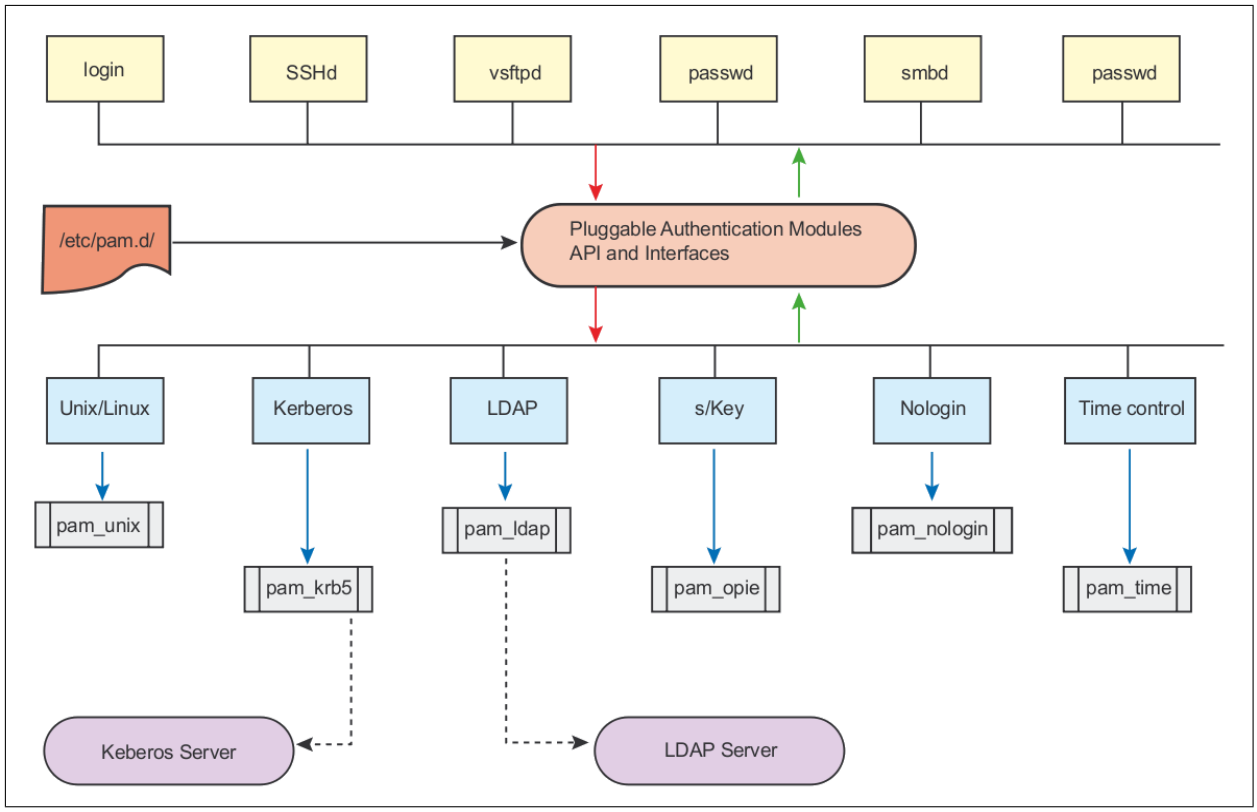

Figure 1: Working scheme of the *Pluggable Authentication Modules* (PAM)

`/etc/pam.d/` includes one separate configuration file for every application which uses PAM. Such a configuration mostly contains lines in the form of:

```
<type> <control> <module> [<arguments>]
```

Where `type` can be

- `account` for non-authentication-based restrictions, e. g. when PAM modules are only allowed to authenticate within a fixed time-slice, independent from who is asking to gain authentication.
- `auth` for the authentication procedure itself (the most important value for this paper).
- `password` for *changing* passwords.
- `session` for special operations which should be executed before or after successful login, e. g. prepare system environments or activate logging facilities.

The `control` field may be one of the following:

- `requisite` makes sure, that the calling application will be refused to authenticate if the selected module couldn't verify the users identification string. If the module exits with failure (`PAM_AUTH_ERR` definition), the whole authentication process is canceled immediately and the requesting application will get a negative response.
- `required` is like `requisite`, but this control flow modifier will wait until *all* other modules of this `type` have been processed (dispite whether successful or not) to return negative response.
- `sufficient` forces to give positive response to the caller if the module has been processed successfully, dispite of return values of other modules.
- `optional` calls modules which can do basically anything but whose return value is being ignored.

Finally comes the name of actual module, namely the shared object file (`.so`) relative to `/lib/security/`, where all PAM modules are stored. A module may introduce a certain number of command line arguments, which may be passed immediately after the module-name. There are basically no standard arguments which all modules would process in the same manner; however, passed but unknown arguments will mostly be ignored.

1.3 A quick introduction to cryptographic hashes

Most standard login procedures force to request

- an identification specifier (e. g. through a username), and a
- verification token (e. g. a password).

There's basically no need for using cryptographic routines until now, as long as the authenticator (the service, for example) knows the exact password string. However, sometimes it's mandatorily necessary that the data which has to be stored by the authenticator is not equal to the password itself and, even more important, doesn't reveal useful information if read by an unauthorized party.

The key for solving this problem is called *hashing*. A cryptographic hash function h takes an (virtually) arbitrarily large input data set D_i of size $|D_i|$ and returns a

fixed-size fingerprint (the *hash*) of the input data. Mathematically spoken, a hash function usually follows a projection $h : \Sigma^* \rightarrow \Sigma'^n$, defining Σ as input alphabet and Σ' as output alphabet.

Hash algorithms deeply rely on some important criteria:

- **Data reduction (data expansion)** Independent from the size of D_i, the corresponding hash value $h(D_i)$ must have a fixed size. The standard case is that $|D_i| > |h(D_i)|$, so the input data is (typically much) larger than the resulting hash value; this behaviour is called data reduction. Otherwise, if $|D_i| < |h(D_i)|$, the resulting hash size is still the same and the data is therefore expanded.

- **Confusion** Similar structures inside D_i shall lead to completely different structures in $h(D_i)$.

- **Collision resistance** The possibility of finding two different data blocks D_1 and D_2 as input, which result in the same hash value $h(D_1) = h(D_2)$ must be minimized. From the logical point of view, true collision avoidance can only be achieved if the output length of the hash algorithm $O = |h(D_x)|$ is at least as large as the length of the input data portion $|D_x|$. However, it must be ensured that in practice, there's no (efficient) possibility of finding such two input data portions.

- **Inexistance of inverse functions** The hash function h must be virtually irreversible. This means it must not be possible to calculate an inverse function h^{-1} which retrieves the original input data D_i out of $h(D_i)$: $D_i \neq h^{-1}(h(D_i))$ with polynomial runtime complexity.

- **Surjectivity** The hash function h shall provide the possibility of reaching virtually every element of the target range, which includes $2^{|h(D_n)|}$ elements, according to which input data D_n has been given (more information on cryptographic keysizes can be found in [LV00]). Besides, the probability of reaching one specific element in the target range should be exactly the same for each element (which describes a uniform distribution). If some elements of the target range could be foreseen as unreachable, the level of security would decrease[3].

[3] Actually, if this is the case, the hash algorithm would probably be considered as *broken* nowadays, due to very high claims of cryptographic routines.

Since it's unthinkable to store clear-text passwords on our authentication devices, the execution flow for writing and verifying authentication information truly sustains upon some clever algorithms where hashing is the key functionality.

I decided to use SHA512, which is standardized in [ips02]. For some deeper inspections from the mathematical point of view, I'd refer to [Ert03], [Sch05] and [Wae03]. A somehow more practical guide can be found in [Pac05].

1.4 Terminology

I'm writing about an *authentication token* or an *authentication storage device* when refering the the physical medium on which our digital authentication information is being written to. This can be seen quite like a keycard. The presence of this device is prerequisite to a successful authentication via this software suite.

By *authentication fingerprints* aka *authentication information* or *authentication hashes* I mean the digital information which is written onto the authentication token. The structure of this information differs in normal and in rescue mode, but this difference doesn't need to be handled separately by the reader, unless s/he itends to modify the existing sourcecode. Detailed information on this can be found in the next chapter.

The software has been written with respect to be as generic and platform-independent as possible (with little changes to be done). Anyway, since it needs PAM and UDEV, it will most likely not be easily compileable and runnable on other systems than GNU/Linux and has not been tested to work with any other operating system until now.

The software has soley been written by the author of this paper and may be redistributed and modified according to the terms of the GNU General Public License [fsf07]. A public project is hosted at Sourceforge, found at `http://usbauth.sf.net`.

2 Security aspects

Since this software is intended to be used for authentication, comprehensive security techniques combined with a secure, proper implementation should have absolute priority. Using ordinary storage hardware for storing authentication information is a little bit tricky and deserves deeper inspection.

I decided to include several techniques with enhancement of security in mind, namely

- a one-time-password engine,
- USB device serial-number verification,
- a special routine for handling so-called rescue devices,
- support for two-factor authentication by providing a PIN.

2.1 One time password engine

If an authentication fingerprint has been successfully read from the authentication token, validity of this information only lasts exactly one login. To reach this, every user/device combination stores an one-time-password sequence number n, which is initialized with value 1. Upon successful authentication, n is incremented by one.

This procedure fulfills several advantages:

- An additional abstraction layer between the real password-information in the configuration file and the information which is finally used for authentication.
- Intrusion detection in case of loss; if an attacker gains access to the token and the machine it has been dedicated to, it's easy to monitor the OTP sequences and check possible suspicious logins since the device has been lost.

The secret initialization vector read from the configuration file (and possibly an additional PIN) is then iteratively hashed $n+1$ times while concatenating the `iv` value to the resulting hash after every iteration, and written back onto the device. The incremented OTP sequence number $n + 1$ itself gets stored on the local filesystem.

The concatenation of the hashing result with the initialization vector after *every* iteration is terribly important. Otherwise an attacker could calculate future authen-

tication fingerprints simply by rehashing values of a stolen authentication device. An descriptive illustration offers figure 3.

The OTP engine commonly disallows using the same authentication information on multiple machines. However, this case is covered by providing authentication fingerprints for every machine the owner of the token has access to.

2.2 Authentication information

Next we'll discuss the information which is written onto the authentication device, which therefore serves as basis for the authentication result. The goal is to find a suitable piece of information, from which

- authenticity according to the user or program which is trying to authenticate can be verified, and which
- doesn't reveal useful information in case of duplication and loss/theft[4].

To achieve this, authentication fingerprints are created based on the following algorithm.

1. Parse configuration file for users and their designated devices
2. Set OTP sequence number n to 1
3. Define a secret initialization vector `iv` which is to be stored in the configuration file
4. If PIN-based authentication is used, read in PIN
5. If PIN-based authentication is used, concatenate PIN string with `iv` value from configuration file, otherwise only use `iv`
6. Hash the resulting string exactly n times (while concatenating the `iv` value to the resulting hash after every iteration)
7. Write the resulting hash onto the device
8. Save the OTP number onto the filesystem specificly for the user/device combination

[4]This includes username and/or user-ID, which isn't stored on the device.

The actual source code for the algorithm above can be found in appendix A.2.

Now we suppose a user or program trying to gain authentication via `pam_auth.so`. What the module gets from PAM is soley the username (the only parameter we will use, at least).

1. Parse configuration file for occurences of the given username
2. Iterate resultset of devices and check if any of those are plugged in
3. When devices have been found, check for the header information
4. If valid header has been found, retrieve all authentication fingerprints
5. If PIN-based authentication is used, read in PIN
6. If PIN-based authentication is used, concatenate `iv` value from configuration file with PIN, otherwise only use `iv`.
7. Get the actual OTP sequence n for user/device association, if present
8. Hash the resulting string of step 6 exactly n times (while concatenating the `iv` value to the resulting hash after every iteration), and check against the authentication information on the device
9. If fingerprints match, increase OTP sequence and rehash resulting string from step 6 $n + 1$ times and write it onto the device [5]
10. Save the new OTP number onto the filesystem

The container layout of how the fingerprints are exactly written on the partition (as well as on rescue devices) is illustrated in figure 2.

This algorithm is derived from the HMAC algorithm, namely hash-based message authentication like described in [KBC97] and related to the operating principle of stream ciphers like RC4.

2.3 USB device verification

Another very interesting topic is about physical dependency. This should be seen as knock-out criteria for token-based authentication, especially when using commodity

[5] Futher version may enhance the performence of this behaviour via single rehashing

storage hardware since otherwise forensic copies of the authentication fingerprints would work with any physical device.

Fortunately USB devices define their own unique USB serial-number which cannot be easily altered. Therefore, when chosing an USB device for storing authentication fingerprints, the serial number is checked every login and authentication will not be successful if the serial number isn't the one specified in the configuration file.

Therefore the verification process strongly depends on the presence of exactly one physical medium which can only be achieved by the PAM module when authencitating through USB devices. Non-USB devices are conceptionally supported by the codebase, but using devices through other interfaces than USB is by no means recommended since physical dependency may not be given, and therefore not intended to be part of this document. Although if certain hardware devices may also provide serial numbers or other valuable information, the read-out procedure often isn't standardized by the communication protocol of the interface itself, but very device-specific. This makes a device-independent lookup very hard to implement.

2.4 Conceptional groundwork for Rescue Devices

For authentication is restricted to only one physical token, the concept of rescue devices becomes interesting. In case a device gets lost or stolen, restrictive PAM configurations may *rely* on the presence of the token for successful authentication. This may result in a problem when the device isn't accessable anymore.

Because of this, special rescue devices can help. A rescue device may optionally be defined in the configuration file, after the definition of the default token. A rescue device is limited to one user only, and only capable of holding exactly one authentication fingerprint.

2.4.1 Untracable data hiding and single-user restrictiveness

The reason for this restriction lies in another security relevant aspect of rescue devices. Normal authentication tokens contain a special header followed by one or multiple authentication fingerprints for one or multiple users, and again followed by a trailer, whereby one single authentication fingerprint is normally stored in hex and therefore as ASCII printable characters.

Rescue devices follow a different concept: The whole partition on which the fingerprint lies is being wiped out and filled with random data when preparing the device. The rescue authentication fingerprint is then stored on the first 64 bytes (representing the 512 bits of SHA512) of the randomized token space in its pure binary form, without previously converting those values to printable ASCII.

This procedure ensures full disputability since it's not possible to prove that the device holds *any* valueable data which could be used for authentication, for the entropy reaches 100% when cryptographically strong hashing is used, and a good random number source is available[6].

2.4.2 Alternative initialization vectorization and rescue PINs

For providing an even higher security, the initialization vector used for the rescue device is not the same as the one used for the default device. This strictly saprates the sources of origin for the authentication fingerprints of both devices. Therefore an extra configuration parameter `riv` has been declared for this purpose. The `riv` parameter is (like `iv`) just any alphanumerical random value without blanks and used for hash initialization.

Moreover, rescue devices must be secured with a PIN which should again be different from the default PIN. Also neither this PIN nor a hash value of it is ever stored anywhere on the token or filesystem. This makes the submission of the correct PIN inescapable. The detailed layout of authentication fingerprints is given in figure 2.

2.5 Implementation-dependent security topics: C vs. managed vs. interpreted languages

The major part of the provided source code is written in pure C. As opposed to new-generation managed languages like Java and C#, C remains as a very low-level language. One of the main differences is the possibility (and mostly compulsion) of direct memory access and pointer arithmetics.

These differences offer much broader capabilities in terms of system-near programming and all-over execution performance. On the other hand the assignment of

[6]This behaviour could still be enhanced by determining a storage-offset value where the actual fingerprint can be found on the rescue medium. This hasn't been implemented yet.

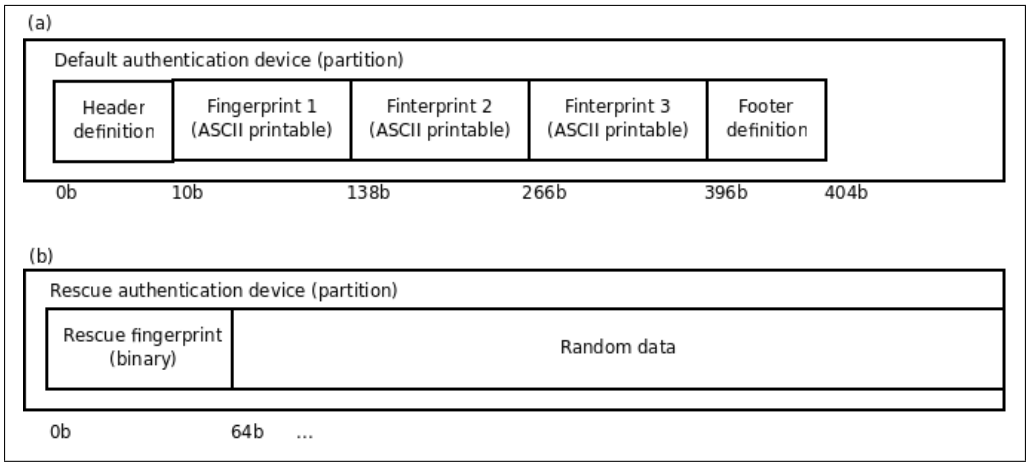

Figure 2: (a) Shows the layout for a standard authentication device: Holding a header, several ASCII-printable 128-byte fingerprints, and a footer. (b) Shows the layout for a rescue authentication device: Holding nothing but a 64-byte binary hash at the beginning of the randomized content.

pointers and therefore direct memory addressing is by far the most serious source of programming bugs and resulting application misbehaviour [Eri08].

Since Linux is inevitably associated with the C programming language, most libraries and therefore programming interfaces are done in C. This fact assumes that pluggable modules (which are realized through `void` function pointers, addressing memory areas in the code segment of the ELF binary or shared object) are either also programmed in C, or a special wrapper is used. As far as I know, the only languages which provide seamless support for programming C-compliant functions are C++ and the new-generation medium-low-level language D. I am currently evaluating the possible benefits of favouring D to C for future work.

Anyhow using C deserves special coding attention to the implementation. Especially the inexistance of a garbage collector offers the possibility of memory leaks, which can turn out to be much more crucial when arising within an external module which has been loaded from another program (in our case, PAM itself), since memory leaks can also force the calling application to crash.

For this reason the memory leak detector `valgrind` has been used throughout the whole implementation process.

In addition, also memory boundries have to be checked seriously since the C compiler doesn't check whether `read` or `write` system call requests point to valid memory areas or not.

Finally, the the C standard POSIX library definitions cover several I/O functions which are commonly treated as insecure due to lacking memory boundary checking. For example, the `gets(char *s)` function writes data from `stdin` to the address where `*s` is pointing to without respect to the actual length of the memory area behind this pointer.

Other such functions are namely `strcat()`, `sprintf()`, `strcpy()` and so on. All of these 'dangerous' functions have been entirely avoided to use.

On the other side of the fence there are *interpreted* languages. Such languages don't use a compiler but an interpreter. Therefore, no machine-dependent code is generated until execution time. This behaviour makes it possible to implement something like a semi-profound managed language, though it's however much slower as according to [Con02].

Perl is an example of an interpreted language, and used for the configuration file generation wizard. Perl unleashes its potency in string processing which is essential for user I/O and file writing. Nevertheless, I wouldn't recommend using interpreted languages for high-security code for Perl is an interpreted language and relies on a Perl Interpreter. This fact introduces another attack vector, since an attacker doesn't necessarily need to attack the Perl code itself, but may focus only on the interpreter. For non-interpreted languages, the execution of binary code is done soley by the Operating System.

2.6 Two-factor authentication (PIN-based verification)

Two-factor authentication is something which doesn't mandatorily have to be done directly via `pam_auth.so`, since the PAM configurations can easily be altered to enforce more than one authentication module per authentication attempt.

Nevertheless, combining `pam_auth.so` with `pam_unix.so` means that (1) the corresponding authentication device has to be plugged in, and (2) the standard Unix password for the particular user has to be given.

It's questionable if this behaviour should be really desirable. On the one hand the usage of `pam_auth.so` will not make life easier (but more secure, however), since all passwords have to be entered as if without having activated this module. Using this combination is still possible without any changes.

On the other hand, introducing one more abstraction layer by implementing the module's own password authentication mechanism would eliminate the need for multiple passwords. It's up to the administrator wheter s/he thinks this is a good idea; I personally think this in general (with reference to *good enough security* as mentioned in [Gar06]).

Moreover, the one password which covers all authentication procedures for the applications which should use this way of authenticating as set by the administrator, is (at least, shoud be) different from all other passwords. This may be seen as advantage, since password stealing attacks (keyloggers, etc.) will have no effect on normal user/system accounts for they are secured with completely different passwords (if any).

Therefore, `pam_auth.so` realizes an optional so-called PIN-based[7] authentication. This means that the authentication process cannot be completed without giving the right PIN. When chosing to use PINs, `uadevwrite` will request the PIN which should be set; this inherently changes the authentication fingerprints which are to be written to the hardware. Once an authentication fingerprint for a specific user has been initialized with PIN-based authentication, verification is mathematically not possible anymore without the right PIN, as shown below.

2.7 Weaknesses

I tried my best to optimize the project in terms of security. But nevertheless, there are some conceptual weaknesses and limitations, which should be mentioned now, and which will most probably be changed in future versions.

- There is currently no support for asymmetric cryptography and therefore no creation or verification support for public/private keys and/or cryptographic certificates.
- The device verification algorithm currently solely allows USB-devices to get securely authenticated, due to the previously mentioned physical dependency.
- The excessive use of UDEV excludes non-Linux operating systems from being used with this project.

[7] A PIN can actually contain up to 64 alphanumeric characters and is not restricted to numbers.

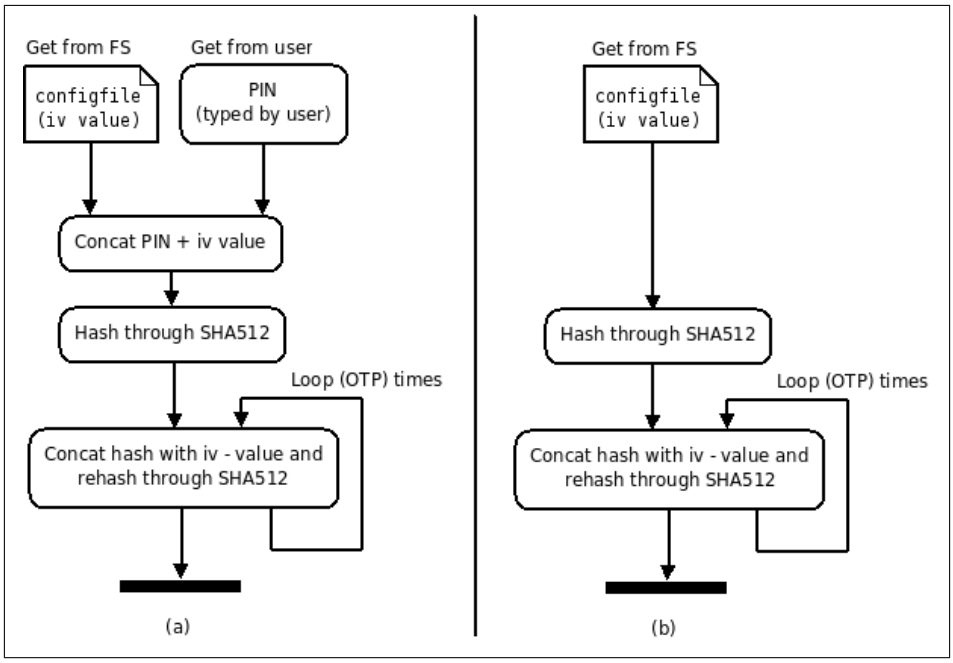

Figure 3: If a PIN is used (a), it must influence the authentication fingerprint which is to be created; this is done by concatenating the PIN with the `iv` value the very first time. If one-factor-authentication is preferred (b), the initialization vector `iv` is the sole source for what's written to the device. Note that the `iv` value is individual to each user *and* each device.

- The untracable-data-hiding procedure isn't capable of being used for non-rescue devices.
- Currently, a dedicated authentication-partition has to be created on the device. This is because it's the easiest way of storing information on a device without existing filesystem. Future versions may suspend this limitation.
- The scripting daemon `uauthd` is still experimental, and listens on local UNIX sockets without strong packet sender authentication. This will make daemon-specific attacks possible, if the attacker has local access to the machine and root privileges[8].
- There is currently no support for a hands-free sign-in. That's because the username of a user a memory device is dedicated to, isn't stored on the device itself. This enhanced security for nobody is able to read usernames off the medium. On the other hand, the verifying system doesn't recognize for whom the device was created for, and therefore manual entering of the desired username is essential.

[8]If an attacker gains root permissions, pam_usbauth won't help your box anyway.

3 Implementation

3.1 Prerequisites

3.1.1 Library linkage and compiler settings

Until now, the OpenSSL library is being linked dynamically within the build process for using hash algorithms. Currently only SHA512 is supported, for I don't see any concrete reason to support multiple hash algorithms. Nevertheless, it's always possible to compile static builds, so the need for actually having OpenSSL libraries installed on the running system fades away.

Using SHA512 routines in source doesn't mandatorily mean that we *rely* on OpenSSL, since it shouldn't be very complicated to bring the SHA512 source directly into the PAM module code. This fact however changes, if we're talking about asymmetric cryptography, for these prodecures require quite a lot of math and encoding standardization. Public key verification is not supported until now, but yet supposed to be implemented in future versions. At latest at this time, the mathematical problems of prime computation and calculation performance suddenly arise and can be obtained in detail from [Rie94] and [AKS04].

Moreover, the `syslog` facility is used for centralized logging through PAM. The PAM handle (which is nothing more than a pointer to a pointer to a PAM structure in C) comes with the authentication request, and allows the call of `pam_syslog()`, without defining an explicit syslog handle, which again reduces error sources[9].

The rest of the used C headers are either POSIX, ANSI, IEEE or OpenGroup standardized and should be available on every Linux box:

[9] Some PAM modules on the net implement own syslog calls and don't free the syslog handles in error cases, which results in PAM segmentation faults (which is *really* not desireable).

`unistd.h`	Definitions for low-level I/O (mainly `read()` and `write()` calls)
`stdio.h`	High-level file I/O
`stdlib.h`	Memory (de-)allocation routines, randomization
`errno.h`	Error numbers
`string.h`	String and memory comparison and manipulation
`signal.h`	POSIX signal definitions
`sys/types.h`	Several Linux structure definitions (e.g. `pid_t` and `pthread` types)
`sys/socket.h`	Generic socket definitions (`struct sockaddr`)
`sys/un.h`	Unix socket definitions (`struct sockaddr_un`)

Compilation has been done with GCC 4.2.3 and POSIX thread model. Compiler code optimization has not been activated for security reasons: Some compilers (including certain GCC versions) enforce code optimization rules by removing any code that attempts to write to addresses in main memory, which are never being read afterwards [Sch07].

However, all memory ranges which are declared to hold sensitive data like passwords, PINs and cryptographic hashes are being wiped out immediately after they're not being used anymore. This should prevent memory leakage, e.g. through `/proc/kcore` image theft.

3.1.2 Favoring UDEV in behalf of DevFS

Since the birth of Unix, devices are getting addressed and used through the `/dev/` directory structure. In early Unix history (and even in early Linux versions), this directory tree had to get built up manually by creating each single node per hand, defining the corresponding major- and minor numbers for special block- or character file creation.

Then came DevFS which represented a device filesystem that magically populated the `/dev/`-tree. DevFS isn't an invention from the Linux party and is heavily used in several operating systems including Mac OS X, Sun (Open-)Solaris, FreeBSD and Plan 9. Beside automatic population of devices into the special filesystem, DevFS brought programs which could be queried to get information about newly connected devices. Nevertheless, in the end it's all about local polling, and therefore not a very efficient way of achieving this.

When I developed an early prototype of the discussed authentication software, it set upon DevFS and the `/proc/` filesystem, namely ProcFS. The latter has been the

sole source for all routines which needed to read out hardware information from the USB subsystem.

Although ProcFS is implemented in a couple of modern operating systems, Linux is the only one putting *way* more information into `/proc/` than what's worth knowing about the sytem's processes themselves. This fact has lead to the introduction of SysFS, populated under `/sys/` in modern GNU/Linux kernels, where fine-grained information about all available hardware may be aquired.

As a matter of fact, also DevFS has been fully replaced in the meanwhile to the benefit of UDEV since GNU/Linux kernel version 2.6 (while DevFS has been completely removed from the kernel sources in 2006). UDEV finally runs in userspace, which offers much more interesting capabilities: Since UDEV automatically monitors, evaluates and interprets hotplug events, this behaviour can be regulated and finetuned by UDEV *rules*, defined in `/etc/udev/rules.d/`. Moreover, UDEV manages an unique device-to-file mapping in opposition to DevFS (and besides, fulfills the Linux-Standard-Base interface definition).

DevFS is out of date while UDEV is quite Linux specific; nevertheless I decided to build upon UDEV since it's the more realistic device filesystem which is to be used in the future. This decision also made the use of `/proc/` obsolete, since all relevant device information may also be found in UDEV handled `/dev/`-trees. Finally, it would make no sense to use DevFS while several hardware parameters have to be read out of `/sys/` which, again, is Linux specific

3.2 Design and architecture

3.2.1 Modularization

The presented authentication suite is divided into a couple of stand-alone modules, logically isolated even from the developer's point of view.

- `libpamauth.so` is a shared object (aka dynamic library) which provides common methods, used by a part of or all other modules.
- `pam_auth.so` is a shared object which represents the actual PAM-compliant module itself. Therefore, PAM must be configured to use this object by changing the PAM configuration files, which we will discuss in detail later.

- `uadevwrite` is a binary which takes the path to a syntactically correct `pam_auth` configuration file, and a device specification (either an USB serial-number or a path to a non-USB device) as arguments. `uadevwrite` will parse the given configuration file and filters entries according to the supplied device on which all determined authentication fingerprints will be written to.

- `uauthd` is a binary which executes daemon-specific code and always runs detached from any console. `uauthd` initially reads out the main configuration file and looks for `[script]`[10] hooks defining actions that should be executed when certain events relevant for authentication are going to happen on the system. This provides a flexible hook-based interface, which makes automatic and transparent script-execution upon authentication events easy.

- `uaconf.pl` is the only program which has not been written in pure C but in Perl. It acts like an initial configuration wizard for creating a fully-functional and syntactically correct configuration file. It guides the user through a step-by-step wizard and finally writes the configuration file to the harddisk, as well as the authentication codes onto the selected device(s). Rescue device support within `uaconf` is currently very limited.

3.2.2 `libpamauth.so`

As this code serves as backend library for all other components, it's discussed first.

Data structues The C data structure `struct t_auth`, which holds all information about exactly one user, his/her authentication storage device, rescue device, initial hashes, one-time-password sequence, PIN and so forth; as well as the structure `t_script`, containing information about parsed `[script]` hooks, the events on which certain scripts are executed, and the association with either an user or a device.

Moreover, the enumerations `enum e_mode` and `enum e_source`, which define the main operating modes `NORMAL` and `RESCUE`, and sources `RAW_DEVICE`, `USBID` and `NONE` respectively.

[10] You may take a look at the Manual section for more precise information on script-definitions

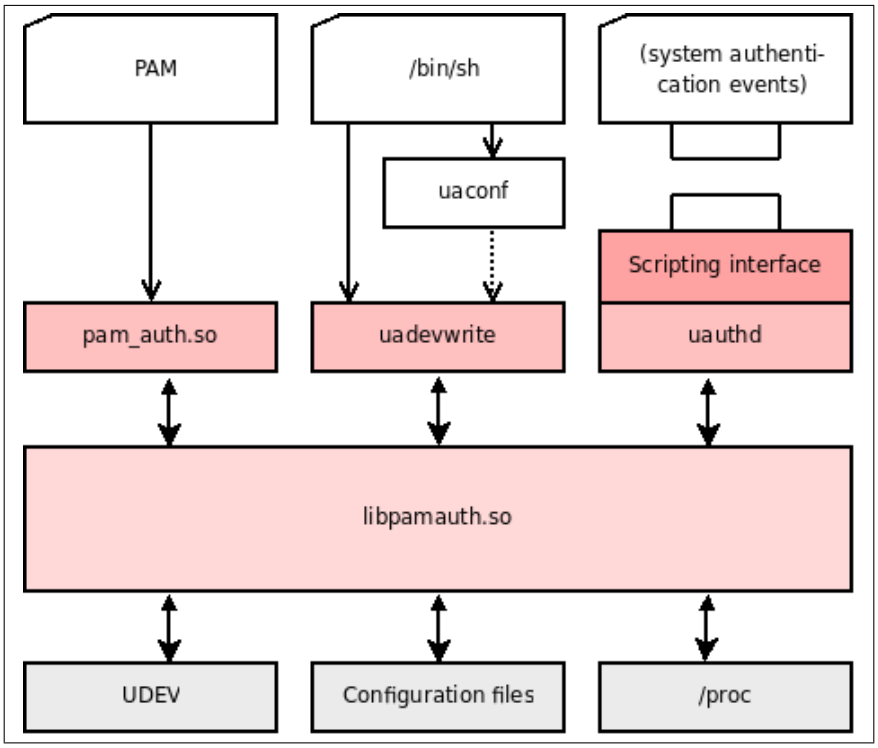

Figure 4: Modularisation, execution flow and involved processes of the authentication suite

Configuration file tokens The syntactical tokens expected to be read out of the main configuration file. This contains the definition of initialization vectors, USB device identification numbers and non-USB-device specifications. Those corresponding C preprocessor definitions may be obtained from appendix B.

Parsing All parsing-specific code, splitted into parsing for `t_auth` structures (`[user] definitions`) and `t_script` structues (`[script]` definitions). Details of the parsing engine are described below.

One-time-password engine (OTP) All functions related to OTP. It currently provides support for reading the current OTP sequence number of a user from the filesystem, writing a new OTP sequence number onto the filesystem, and rehashing string sequences in order to calculate a new one time password after successful authentication. The source code of the rehashing procedure is shown in appendix A.1.

Device writing The code which handles write routines onto an authentication device in correct place, order and syntax.

3.2.3 `uadevwrite`

`uadevwrite` takes a configuration file, searches for a specifically given device or USB serial-number and writes authentication fingerprints of all users found within the given configuration onto the device. Since raw write access is needed on the storage medium, `root` permissions are prerequisite.

The first thing this code does, is calling `libpamauth.so` to parse the configuration file which is given as the first command line argument. The backend library will now fill an array of `t_auth` structs, one for every user found in the config.

Next, it forces to lookup the device, which is given as the second and last command line argument. This can be an USB serial number or a simple path to a storage device via a link to a `/dev/` pseudo-file. Once the device has been successfully found, it's immediately opened for writing. As long as we're in non-rescue mode, the token designation header is being written onto the beginning of the partition which has been declared to be used for storing authentication information.

In non-rescue mode, all users parsed from the configuration file are iterated now. If a user definition deserves an additional PIN/password authentication, the user gets asked to give a PIN through utilizing PAM to do this job.

Per user, his/her one-time-password number n is initialized with 1. The hashing-engine of `libpamauth.so` will now iteratively rehash the result of a string concatenation of the initialization vector from the configuration file and the given PIN exactly n times. The resulting hashed string is the authentication token which will be written onto the storage device after converting hex, followed by a newline character '`\n`'.

Once all codes have been successfully written, the authentication space is closed by the token designation trailer as shown in figure 2.

When writing fingerprints on rescue devices, the first `[user]` occurence who has the given USB serial number (or device) configured as rescue device will be used for writing. Since a rescue device may only contain exactly one fingerprint, that's ok so far. `uadevwrite` will ask for a specific rescue passphrase if this is the case – which should by no means correlate with the non-rescue PIN, if any.

3.2.4 uauthd

The uauthd daemon implements the event and scripting interface. When the program starts, it forks, kills it's parent, detaches from console, closes the stdin, stderr and stdout pseudo-filedescriptors and open a syslog channel for event logging.

Signal handlers for SIGTERM and SIGHUP are being registered. On SIGTERM, the main configuration file is being reread, which leads to a rebuild of the t_script array. SIGTERM removes previously created UDEV rules, if any, and exits.

Finally, a Unix socket is registered and bound, and is ready for listening and accepting connections. If an event is being received, it's caught by the message-handle function which decodes the message string, looks up the received parameters in the script configuration map and executes all scripts which have an execution definition of the event-type happened.

The reason for designing the event-interface via sockets is extendability and scalability: Future version may introduce numerous types of events and authentication structures, which will all be handled in a centralized way through the daemon. Moreover, a more enterprise-oriented scenario would include execution of certain scripts on remote servers (for logging, accounting, etc.) – to modify the code to fulfill this demand, the UNIX sockets have just to be replaced by standard stream sockets (like TCP).

If a device-based event is getting caught (e. g. device plugging and unplugging), the actions which are to be executed are stated within an UDEV rule, before UDEV is getting restarted by calling `# udevcontrol reload_rules`. The rule created will be called `01-pamauth.rules` and can be found in `/etc/udev/rules.d/`.

3.3 Parsing

The main configuration file syntax has been designed for holding exactly one independent user or script definition per line, while the hash character # serves as comment-indicator.

Therefore the configuration file is being read line by line, while lines beginning with # or a newline (\n) are immediately skipped.

Depedent on what the caller intends to get parsed, only lines with [user] or

`[script]` keywords are being processed. Correspondingly, arrays of `t_auth` or `t_script` are filled with all available, syntactically correct information from the given configuration source. All fields of the structures are being wiped out, except what's not found within the configuration (like PINs and OTP sequences). Note that PINs are never ever stored anywhere, neither the hosts filesystem nor the authentication device.

3.4 Multi-user/multi-token cross-validation

Early prototypes of `pam_auth.so` only supported one user fingerprint on exactly one device. This worked fine for single-user hosts, but isn't manageable anymore in some wider context and therefore lacks scalability.

This fact made me implement a multi-user multi-token cross-validation algorithm. Hence, one physical token is capable of holding an unlimited[11] number of user authentication fingerprints, while one user can have his/her fingerprints stored on as many tokens s/he likes, without interfering. One-time-password sequences are stored for every user/token relation, which in consequence creates a logical separation from different tokenspaces.

As a result, a user may e.g. have two different physical tokens, whereby

- device 1 stores authentication information for logins, and
- device 2 stores authentication information for key decryption routines.

If a device gets stolen, this has absolutely no effect on using the second device.

Practically, the network's administrator may also create an image of a token which stores rescue information for *all* users of his/her network (each on a separate partition), and encrypt it for security reasons, which is a useable fallback-scenario. Figure 5 shows the validation concept.

3.5 Generic scripting interface

The `uauthd` daemon provides a powerful event-based hookable scripting interface. The scripts and commands which are to be executed on certain events are again

[11] Well, of course it's limited by the amount of space on the device. But this is the job of the administrator.

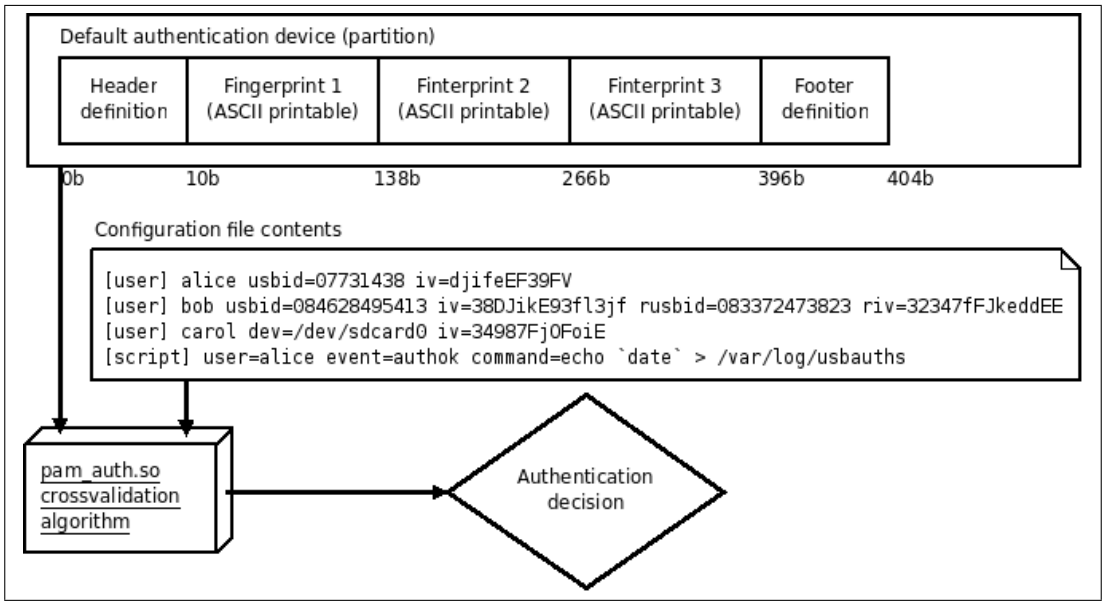

Figure 5: `pam_auth.so` performs cross-validation across what's defined in the configuration file, and what have been found on the authentication token.

defined in the main configuration file. The design and implementation of `uauthd` has been done according to be easily extendable for additional events in the future (e.g. more detailed identity information; a concept of smart events, looking for possible attacks, etc.).

The implementation currently supports *proactive* and *reactive* event processing:

- Proactive event processing is associated with certain devices. For example, if the administrator wants `uauthd` to execute code when a given device has been plugged in, this event is completely independent of any authentication procedure, and therefore PAM. On daemon initialization, special UDEV rules are created for executing the scripts defined in the configuration file, which are deleted on receiving the SIGTERM signal.

- Reactive event processing is associated with user-specific events, like successful or unsuccessful authentication through PAM. To achieve this, `uauthd` implements a generic UNIX socket server which listens for incoming messages. Those will most likely come from `pam_auth.so`, but also other applications (running with root privileges) can build event messages and send them to the daemon. The daemon will process the message, extract all necessary information out of it and handle the hook by what's defined to do upon occurence of this event. Because of running out of time, the current version of the server doesn't use an authentication protocol for the messages. Therefore, it's not recommended

yet to use the socket-related scripting fuctions in high security environments.

The implementation of this event-based message-queue has been done quite generic, so it won't take more than a few minutes to port the Unix socket code to TCP/IP sockets (by using `sockaddr_in` instead of using `sockaddr_un`). The reason for chosing Unix sockets in favour of TCP/IP sockets is motivated by security reasons, since Unix sockets are per definition limited to the local machine, which eliminates the need for specifically locking out all other requests from the network, and is therefore less vulnerable.

3.6 Achieving filesystem independence and retaining storage media operationality

Filesystem independence in this context means independence of which filesystems the administrator wants to use the authentication storage device with. The authentication fingerprints are stored (by default) at the logical end of the device' space, while using just a few kilobytes of data.

However, the exact location where the 'sensitive' information is being stored, isn't fixed. It's possible to store that data portion virtually everywhere on the device, but the configuration file generation wizard `uaconf` is designed to create a partitiontable which reserves only the last megabyte chunk of data for us.

This is essential, since Microsoft Windows is only capable of handling exactly one partition on USB flash drives, which is the first one. All others are ignored; therefore, the first partition on the device is being created with the all-over size of the device, subtracted by 1024kb, and formated with FAT32 by default. The last kilobytes are subsummed to an autonomous parition and used for storing authenticating information. This partition doesn't contain *any* filesystem, so the authentication mechanism cannot be limited by any (possibly missing, if thinking about embedded Linux devices) filesystem driver as long as raw write support is given (which must be present anyhow, to use the device).

With this technique, the authentication tokens can be used like normal storage media drives without interfering with sensitive authentication data, to nearly its full extent in terms of usable disk space. This means you don't actually have to carry an extra *authentication token* with you, but just your day-to-day flash drive – only with special information being stored upon.

4 User manual

4.1 Compiling and installing

Unix-based systems often rely on compiling and installing software packages manually before using. That's why I decided to include these steps in the user manual.

Before compiling, you have to make sure that the OpenSSL and PAM development packages are installed, including the libraries themselves as well as the C header files. Of course, also some tools for the build-process such as `make` must be present. On Debian and some Debian-derived GNU/Linux distributions like Ubuntu, this is done by typing

```
# apt-get install libpam0g-dev libpam0g libssl libssl-dev build-essential.
```

Now the sourcecode of the software package is ready to get checked out. A subversion repository is available at `delta-xi.net`, anonymous checkouts are allowed. This operation requires the subversion client to be installed.

```
$ svn co svn://delta-xi.net/svn/pam_auth-ng
```

Unless you don't actually want to change the inner behaviour of the software, nothing more needs to be configured at this point, we're ready to procees with compilation; afterwards installation can follow.

```
$ make && sudo make install
```

The installation process will install the following files:

- `pam_auth.so` into `/lib/security/`, where all other PAM modules are installed.
- `uawritedev` into `/usr/sbin`, to the superuser binaries.
- `uaconf` into `/usr/sbin`.
- `uauthd` into `/usr/sbin`.
- A sample configuration file `pam_auth-ng.conf` into `/etc/`.
- Manpages for all binaries into `/usr/share/man`[12].

[12] Not all of them have been finished, though.

4.2 Using the uaconf configuration wizard

uaconf is a Perl script which acts as configuration wizard for easy configuration file generation as well as device-preparation application. The tool has been written primarily to create configurations for USB devices with USB serial number verification; non-USB devices are not supported so far. Anyway, I don't even recommend to use anything other than USB devices for security reasons. The program must be started as root. The authentication hardware's partition-table is being prepared to hold the fingerprints written by uadevwrite.

```
$ uaconf
Sorry, you have to be root.
$ sudo uaconf
Make sure that your main authentication USB device is plugged in, then press enter.
(and in addition, if you like, a second device as backup-solution).
```

uaconf tells us to make sure, that the USB device which we want to use is plugged in.

```
Searching for USB storage devices... OK
  [0] GH_PicoBit pointing to /dev/sdb [USBID 0773143900A6], 1029MB
  [1] HITACHI_USB pointing to /dev/sdc [USBID 0343948858477], 4096MB

Please enter the number of the device you want set up for use with usbauth: 0
```

The tool performs a scan over all available USB host controllers and filters mass storage devices. We are now asked to enter the number of the device we'd like to prepare as authentication token.

```
The chosen device contains the following currently mounted partitions:
  [partition] /dev/sdb1

Try to unmount? (Y/n)
```

If the selected device is currently mounted, you're be asked if it should be unmounted. The wizard will exit if unmounting is not possible; however, since UDEV is required for running the software and therefore in use at this point, errors shouldn't happen. Default value is Yes.

```
  Trying to unmount /dev/sdb1
Ok, all partitions have been unmounted. Let's proceed.


************************************************************
                        W A R N I N G
************************************************************
All data on the selected device will be lost!
Are you -sure- to prepare the USB memory device
    GH_PicoBit [USB ID: 0773143900A6],
    located at /dev/sdb
    holding 1029MB of data? (y/N)
```

That's it for our turn. `uaconf` has all information it needs to proceed and finish the device preparation. The next step is to wipe out the whole device and create a new partitiontable structre onto the medium; this is a quite sensible part, so we're once more asked to verify what we have chosen so far. If the actual device selection points to an (USB-) harddisk, data will possibly be irreversibly lost.

```
Deleting partition table... OK
Creating new partition table... OK
Everything seems to be fine.
Your config file has been written to pam_auth-ng.1.conf
```

Device preparation has finished. A minimal configuration file for exactly one user on one device has been written into the file `pam_auth-ng.1.conf`, as we were told. Now that we have a well-prepared storage device and a hopefully working minimal configuration file, we're ready to proceed to the next step: writing the initial authentication fingerprints onto the device with `uadevwrite`.

4.3 Using the `uadevwrite` tool for device activation

Finally the authentication information has to be written onto the storage device. `uadevwrite` is a simple tool which reads a given configuration file (argument one), searches for a specifically given device (argument two) and writes all hereby associated user fingerprints onto this device (if plugged in, of course). This is quite straight, no syntactic sugar:

```
$ sudo uadevwrite ./pam_auth-ng.1.conf 0773143900A6
  Writing data on 0773143900A6 for user root with password djaf83po38djr...
     PIN:
```

For each user whose authentication information has been written to the device, one line is stated. If two-factor authentication has been enabled (which is the case by default, when creating a configuration by using the wizard), `uadevwrite` will ask for a *personal identification number* (PIN) which acts as device decryption-password. This PIN is never ever stored, neither locally nor on the device.

There is one more feature of `uadevwrite`, namely to write rescue authentication fingerprints onto a rescue device. This time, the second argument is the USB serial number (or non-USB standard storage devicefile) of the device which should act as rescue device for a specific user. This device number has to be explicitly configured in the configuration file (see the `rusbid=` entry in the configuration file section).

Rescue mode is activated by adding a third argument `rescue` to `uadevwrite`. **Note:** Since a rescue device is only capable of holding exactly one authentication token, only the very first appearence of this USBID (or devicename) as rescue device in the configuration file will be used, all forthcoming entries are ignored. The same is true for `pam_auth.so`, which performes the rescue device check.

Because of a special data hiding procedure, the restriction[13] of holding only one fingerprint has to be accepted due to a more secure environment. The final call would look just like this:

```
sudo uadevwrite ./pam_auth-ng.1.conf 0773143900A6 rescue
```

4.4 Using the `uaauthd` daemon

The usage of `uaauthd` is optional. The daemon will enable the scripting and event interface and is solely configured through the main configuration file by `[script]` definitions.

As events are caught, the execution of scripts will be done according to the actual configuration. See the next section for detailed syntactical analysis of the configuration file.

[13] Altough I wouldn't call this a restriction or limitation, is basically just a rule.

If the daemon doesn't run, `[script]` lines are just ignored and thrown events from `pam_auth.so` are simply terminated by the Unix socket stack. To run the daemon, simply start the service by typing:

```
# uauthd
```

The daemon forks, kills it's parent, detaches from console, closes `stdin`, `stderr` and `stdout` descriptors and open a `syslog` channel for event logging.

4.5 Understanding the main configuration file

The main configuration file is usually stored in `/etc/pamauth-ng.conf`. All lines beginning with a hash '`#`' or a newline character are skipped. This isn't the case for hashes inbetween lines for they play an important role on partition number designation.

4.5.1 `[user]` definitions

Every line holds exactly one definition of a user, his/her authentication device, and an optional rescue authentication device. The syntactical structure on such a user definition follows the regular expression

```
\[user\] USERNAME usbid=USBID#PARTNUM[+] iv=RANDSTR [rusbid=USBID#PARTNUM riv=RANDS
```

whereby the mentioned fields are defined like this:

- `USERNAME` Loginname of the user which the authentication device belongs to (e. g. the user which is allowed to gain authentication via the given device).
- `USBID` Serial number identification (USB device ID) of an USB device. The serial numbers of USB devices can be read out via `/dev/disk/by-id/`, as well as via `/sys/`, `lsusb` and `lshw` (and additionally via `/proc/` on some older Linux versions).
- `PARTNUM` A single-digit number of the partition on which the authentication fingerprints are being stored. According to Linux `/dev/[h|s]dX` standard, 1 is associated with the first partition. A trailing '`+`' character indicates that this USB device is marked as PIN-secured and will force PAM to request a password.

- `RANDSTR` This may be an alphanumerical random string without blanks. `pam_auth.so` will use this as initialization vector for the one-time-password engine. This value doesn't have to be remembered like a password, it should just be a random string.

When PIN-based authentication is used (remember the trailing '+' character after the partition number definition), the fingerprints which are to be read from the device, are presalted with the password initially given at device preparation. Therefore, removing this mark after it has been created in PIN-enabled mode, will prevent logins. This behaviour is considered as security feature for configuration file modification attacks[14].

The syntax may be caught easier on sample inspection:

- `[user] alice usbid=93738343#2 iv=FJDj38f90f`
  This line will allow user `alice` to authenticate via the USB device identified by the serial number 9373834. Authentication tokens will be stored on the second partition. (Note: No PIN-based security, therefore only single-factor authentication. No rescue device specified)
- `[user] alice usbid=93738343#2 iv=FJDj38f90f rusbid=23647384#2 riv=ffeiodjF38d`
  In addition, Alice wants to write rescue-information on another device (identified by serial number 23647384, authentication fingerprints found on the second partition).
- `[user] alice usbid=93738343#2+ iv=FJDj38f90f`
  This time, again no rescue device is specified, but PIN-based authentiation has been activated. This enables two-factor authentication.

Therefore, a rescue device is bound to one user solely.

4.5.2 `[script]` definitions

Every line holds exactly one definition of a script and under which circumstances (meaning upon which events) it should be run.

The syntax is the following:

[14] Your security policies will most likely suffer from larger problems, if attackers are able to read or write to the main configuration file.

```
[script] user=USERNAME event=authok|autherr command=COMMAND
[script] usbid=USBID event=plugin|plugout command=COMMAND
```

When associating a script with events corresponding to a specific user, the first line is what we need. The currently supported user-based events are `authok` for successful authentication of user `USERNAME`, and `autherr` on authentication failure.

When associating a script with events according to a certain USB serial number, the second line shows us the way. The currently supported device-based events are `plugin` when the device is firstly recognized, and `plugout` on device unplugging.

And finally, the `COMMAND` string defines the exact path to the binary or script which shall be executed, including its argument list (without quote signs).

- `[script] user=alice event=authok command=/usr/bin/touch /timestamp`
  This sample line defines a script, which is to be executed when user `alice` authenticates successfully.
- `[script] usbid=077314380188 event=plugin command=/usr/bin/touch /timestamp`
  In opposition, this line binds the event to a specific USB serial number, and the command string gets executed when the device is plugged in (not depending on any authentication). The device doesn't even deserve to get recognized as a storage medium (USB mass storage clas, 0x8).

To avoid misconception, if a `[script] user=`-line is specified, the command string will only be executed when authenticating via the `pam_auth.so` PAM module, and not on other authentication methods.

4.6 Preconfiguring PAM

The Pluggable Authentication Modules architecture must be told to actually use `pam_auth.so` for particular applications. Hereby, PAM defines a separate file inside `/etc/pam.d/` for every PAM-capable application. By default, most of these applications include the file `/etc/pam.d/auth-common` which defines authentication via `pam_unix.so`, the standard Unix user authentication.

To use our storage device module, the easiest way is to add the line

```
auth sufficient pam_auth.so
```

to this file. Alternatively, this line can also be added to certain applications only, the device-based authentication should be used for. For example, if we want to enable `pam_auth.so` for the `su` command the file `/etc/pam.d/su` may look something like this:

```
# The standard Unix authentication modules, used with
# NIS (man nsswitch) as well as normal /etc/passwd and
# /etc/shadow entries.
# @include common-auth
auth required pam_auth.so

@include common-account
@include common-session
```

If `@include common-auth` is uncommented, *both* authentication mechanisms will be executed (`pam_unix.so` and `pam_auth.so`).

5 Outlook and conclusion

The introduced implementation of an authentication procedure through ordinary storage devices is somehow a proof-of-concept reference to this model. There are many features which could still be extensively enhanced, particularly

- device-specific symmetric cryptographic routines for en- and decrypting the whole data (including header, fingerprints and trailer) required for successful authentication,
- support for authentication through asymmetric cryptography smartly using public/private keypairs,
- support for medium-specific authentication fields (the USB serial number is by far not the only information stored on USB hardware; just take a look into `/sys/bus/usb/devices/`),
- support for storage media verification for non-USB devices,
- multitoken authentication (e. g. two different tokens must be present to enable successful authentication; this may make sense if a special service may only be used if two persons simultaniously plug in their authentication tokens), etc.

Also porting the software to other PAM-compliant operating systems like Sun Solaris/OpenSolaris seems interesting. Hereby, a device filesystem independent access model would have to be introduced in form of an additional abstraction layer, where UDEV support embodies only one option of accessing raw devices.

A Source code excerpts

A.1 The OTP rehashing procedure

The following piece of code is part of the `libpamauth.so` backend and therefore used in all other modules. It takes a block of data which is the be hashed, the number of rehashing iterations to be accomplished, and an optional salt value. The salt is used if PIN-based authentication has been chosen, therefore the PIN itself serves as salt.

```
1  char* otp_rehash( char *tobehashed, int iterations, char* salt ){
2    int i, j;
3
4    /* resulting 512bit hash from SHA512() */
5    unsigned char* sha512_bin =    (unsigned char*)
6      malloc(sizeof(unsigned char) * (SHA512_DIGEST_LENGTH + 1));
7
8    /* ASCII string of 512bit-binary hash in hex (doubles length) */
9    char* sha512_ascii =  (char*) malloc(sizeof(char) *
10             (SHA512_DIGEST_LENGTH*2 + 1) );
11
12   /* Buffer which temporarily holds (hash in ascii)+(IV in ascii)*/
13   char* rehash_buffer = (char*) malloc(sizeof(char) *
14             (SHA512_DIGEST_LENGTH*2 + 1)*2 );
15   char* p;
16
17   if( rehash_buffer == NULL ||
18     sha512_bin == NULL ||
19     sha512_ascii == NULL) {
20     syslog(LOG_WARNING, "(otp_rehash) malloc() error\n");
21
22     return NULL;
23   }
24
25   /* zero out rehash buffer */
26   memset(rehash_buffer, 0, (SHA512_DIGEST_LENGTH*2 + 1)*2 );
27
28   if ( iterations < 1 )
29     return NULL;
30
31   /* rehash config-file-hash user->otp_sequence times */
32   for(i = 0; i < iterations; i++){
33     /* initial hash for first iteration comes from config file,
34       then rehashing is done via previous hash */
35     strcpy( rehash_buffer, (i) ? sha512_ascii : tobehashed);
36
37     /* n iterations, concatenate IV from config every time (HMAC) */
```

```
38      if ( i )
39        strcat( rehash_buffer, tobehashed );
40      else
41        if ( salt != NULL)
42          strcat( rehash_buffer, salt );
43
44      /* zero buffers */
45      memset(sha512_bin, 0, SHA512_DIGEST_LENGTH + 1);
46      memset(sha512_ascii, 0, SHA512_DIGEST_LENGTH*2 + 1);
47
48      /* Get real SHA512 hash from OpenSSL */
49      SHA512( (const unsigned char*) rehash_buffer,
50            strlen(rehash_buffer),
51            (unsigned char*) sha512_bin );
52
53      /* convert binary SHA512 output to printable ascii */
54      for (p = sha512_ascii, j=0; j < SHA512_DIGEST_LENGTH; j++) {
55        *p++ = "0123456789ABCDEF"[(sha512_bin[j] >> 4) & 0xf];
56        *p++ = "0123456789ABCDEF"[sha512_bin[j] & 0xf];
57      }
58      *p = '\0';
59
60      /* zero out the buffer; prevent /dev/kcore dump attacks */
61      memset(rehash_buffer, 0, SHA512_DIGEST_LENGTH*2 + 1);
62    }
63
64    free(rehash_buffer);
65    free(sha512_bin);
66
67    return sha512_ascii;
68  }
```

A.2 The device activation procedure (as done by `uadevwrite`)

The following function is part of the execution flow of the `uadevwrite` program. It parses the configuration file, opens the given devices (if present) and writes the corresponding authentication information onto the device. The procedure for creating rescue devices is also included in this piece of code, which is quite different as stated above.

```
1  int main ( int argc, char ** argv ) {
2    FILE *fd_device = NULL;
3    enum e_mode mode = NORMAL;
4    int max_users = 50, i, num_parsed_users, hashes_written = 0;
5    char sha512bin_rescue[SHA512_DIGEST_LENGTH*2+1];
6    struct t_auth users[max_users];
7    char rescue_pin[MAX_FIELD_LEN], buf[MAX_HASH_LEN+MAX_FIELD_LEN+1];
8
```

```
9   perform_prechecks( argc, &mode );
10
11  /* Parser config file + fill users[] array */
12  num_parsed_users = get_users_from_cfg( users, argv[1], max_users );
13  if ( num_parsed_users == 0 )
14    fatal("Error: Could not obtain valid data from config file.\n");
15
16  if ( debug == 1 )
17    printf("Loaded %d users from configfile.\n", num_parsed_users );
18
19
20  i = lookup_given_device( argv, users, num_parsed_users, mode );
21
22  /* open device given as argument, and write header  */
23  fd_device = fopen( (mode==NORMAL)
24    ? users[i].token_dev
25    : users[i].rescue_dev, "w" );
26  if ( fd_device == NULL )
27    fatal("Couldn't open specified device. Did you plug it in?\n");
28
29  /* write head to device, if in normal mode */
30  if(mode==NORMAL)
31    fwrite( DEVICE_HEADER, strlen(DEVICE_HEADER), 1, fd_device );
32
33
34  /* iterate users[] array, write all usertoken from config */
35  for ( i = 0; i < num_parsed_users; i++ ) {
36    /* does current user define given serial/device? */
37    if( !check_dev(mode, users, i, argv[2]) )
38      continue;
39
40    /* print data */
41    printf( "\tWriting %sdata on %s for user %s with password %s... ",
42      (mode==NORMAL) ? "" : "RESCUE ", argv[2], users[i].username,
43                (mode==NORMAL)
44                  ? users[i].hash_cfg
45                  : users[i].hash_cfg_rescue );
46
47    /* if in RESCUE mode, write rescue-auth-structure and exit */
48    if (mode == RESCUE) {
49      /* write to rescue device:
50        [sha512(cfghash)][sha512(cfghash+rescue_pin)] */
51      printf("\n\t\tEnter PUK: ");
52      fgets( rescue_pin, MAX_FIELD_LEN, stdin );
53      rescue_pin[strlen(rescue_pin)-1] = '\0';
54
55      SHA512( (const unsigned char*) users[i].hash_cfg_rescue,
56        strlen(users[i].hash_cfg_rescue),
57        (unsigned char*) sha512bin_rescue );
```

```
58        snprintf(buf, MAX_FIELD_LEN+MAX_HASH_LEN, "%s%s",
59          users[i].hash_cfg_rescue,
60          rescue_pin);
61
62        SHA512( (const unsigned char*) buf,
63          strlen(buf),
64          (unsigned char*) sha512bin_rescue+SHA512_DIGEST_LENGTH );
65
66        fwrite( sha512bin_rescue, SHA512_DIGEST_LENGTH, 2, fd_device );
67        fwrite( "\n", 1, 1, fd_device );
68        fclose(fd_device);
69
70        /* exit immediately */
71        return EXIT_SUCCESS;
72      }
73
74      /* Entering NORMAL mode (write normal auth-tokens on given device)*/
75      /* Get PIN if requested */
76      if (users[i].use_pin) {
77        printf("\n\t\t%s", MSG_ENTER_PIN);
78        fgets( users[i].pin, MAX_FIELD_LEN, stdin );
79
80        /* terminate with \0 instead of \n */
81        if ( users[i].pin[ strlen(users[i].pin)-1 ] == '\n' )
82          users[i].pin[ strlen(users[i].pin)-1 ] = '\0';
83      }
84
85      /* only hash once; if PIN given, use as salt */
86      write_hash_on_device ( users[i].hash_cfg, 1, fd_device,
87        (users[i].use_pin)
88          ? users[i].pin
89          : NULL )
90      ? printf("failed [!!]\n")
91          : printf("done\n");
92
93      /* save otp number */
94      if ( !set_otp_on_fs( users[i].username,
95          (users[i].token_dev_type == RAW_DEVICE)
96            ? users[i].token_dev
97            : users[i].token_usbid,
98          users[i].token_dev_type, 1 ))
99        printf("\tError: Could not create OTP file. Going on...\n");
100
101     if ( debug == 2)
102       printf("[DEBUG] otp_sequence = %d\n", users[i].otp_sequence);
103
104
105     /* initialize user struct */
106     init_user_struct (&users[i], NULL, USER_ATTR_NONE, USER_ATTR_NONE);
```

```
107       hashes_written++;
108     }
109
110     if ( !hashes_written )
111       printf("Serial␣%s␣is␣not␣defined␣in␣%s\n", argv[2], argv[1]);
112
113     /* write footer, close file, if in normal mode*/
114     fwrite( DEVICE_FOOTER, strlen(DEVICE_FOOTER), 1, fd_device );
115     fclose(fd_device);
116
117     return EXIT_SUCCESS;
118 }
```

B Noteworthy C Preprocessor definitions

The `libpamauth.so` backend library defines a couple of interesting C preprocessor definitions, which may possibly be altered according to the currently running system. This allows some measure of tuning the PAM module.

Definition of the authentication token header and footer designations, which are used to identify (normal) authentication devices. If this information isn't given, the device will be treated as invalid.

```
1 #define DEVICE_HEADER                 "<pam_auth-ng>\n"
2 #define DEVICE_FOOTER                 "</pam_auth-ng>\n"
```

Path definitions, including the full paths to the configuration file, the storage directory for one-time-password sequence numbers, the Unix socket path, the UDEV rules directory and the `udevcontrol` binary for reloading UDEV rules.

```
1 #define AUTH_CONFIG                   "/etc/pamauth-ng.conf"
2 #define AUTH_OTP                      "/etc/.pamauth.otp/"
3 #define UNIX_SOCK_PATH                "/tmp/.ua.sock"
4 #define UDEV_RULE                     "/etc/udev/rules.d/01-pamauth.rules"
5 #define UDEV_CONTROL_RELOAD           "/sbin/udevcontrol␣reload_rules"
```

Definition of maximum buffer values. If for example (much) longer initialisation vectors are requred it's possibly needful to increase the `MAX_READ` value. All other definitions should be self-explanatory.

```
1 #define MAX_READ                      512
2 #define MAX_FILENAME_LEN              512
3 #define MAX_FIELD_LEN                 128
4 #define MAX_HASH_LEN                  200
5 #define MAX_DEVICE_BUF                2048
6 #define MAX_SCRIPT_SIZE               50
```

Definitions for the configuration file attribute preembles. The first definition set holds the syntactical token list for `[user]` lines, the second set the `[script]` lines.

```
1  #define CFG_IV                    "iv="
2  #define CFG_USBID                 "usbid="
3  #define CFG_DEV                   "dev="
4  #define CFG_RIV                   "riv="
5  #define CFG_RUSBID                "rusbid="
6  #define CFG_RDEV                  "rdev="
7
8  #define CFG_USER                  "user="
9  #define CFG_SERIAL                "usbid="
10 #define CFG_EVENT                 "event="
11 #define CFG_CMD                   "command="
```

References

[AKS04] M. Agrawal, N. Kayal, and N. Saxena. Primes is in p. Department of Computer Science and Engineering, Institute of Technology Kanpur, India, 2004. (URL http://www.cse.iitk.ac.in/users/manindra/algebra/primality.pdf).

[Bau05] Michael Bauer. *Sichere Server mit Linux*. O'Reilly, 2005.

[Con02] Michael Connell. Pyhton vs. Perl vs. Java vs. C++ runtimes, 2002. (URL: http://furryland.org/ mikec/bench/).

[Eri08] Jon Erickson. *The art of exploitation, 2nd edition*. No starch press, 2008. (ISBN: 1593271441).

[Ert03] Wolfgang Ertel. *Angewandte Kryptographie*. Hanser Fachbuchverlag, 2003.

[fsf07] The free software foundation. Gnu general public license version 3. Free software foundation, 2007.

[Gar06] Cara Garretson. Gartner IT security summit: Good enough security may just be fine. Gartner IT Inc., 2006.

[Hil07] Brad Hill. Attacking web services security. iSEC partners, 2007.

[ips02] Federal information processing standards. Specifications for the secure hash standard, 2002. (URL: http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf).

[KBC97] H. Krawczyk, M. Bellare, and R. Canetti. Rfc 2104: Hmac: Keyhashing for message authentication, 1997. (URL: http://www.ietf.org/rfc/rfc2104.txt).

[LV00] Arjen K. Lenstra and Eric R. Verheul. Selecting cryptographic key sizes. Citibank N.A., Techn. Universiteit Eindhoven, 2000.

[Pac05] Lars Packschies. *Praktische Kryptographie unter Linux*. Open source press, 2005.

[PC04] Cyrus Peikari and Anton Chuvakin. *Security Warrior*. O'Reilly, 2004.

[PW06] Johannes Plötner and Steffen Wendzel. *Netzwerksicherheit*. Galileo press, 2006.

[RC00] Ryan Russel and Stace Cunningham. *Hacking exposed*. Osborn Media, 2000.

[Rie94] H. Riesel. *Prime numbers and computer methods for facorization*. Birkhaeuser, 1994.

[Sch98] Reinhard Schmidt. Skriptum zur Vorlesung Kryptologie, Hochschule Esslingen, 1998. (URL: http://www.it.fht-esslingen.de/ schmidt/vorlesungen/kryptologie/seminar/es9798/html/prim/).

[Sch04] Bruce Schneier. *Secrets and Lies. IT-Sicherheit in einer vernetzten Welt.* Dpunkt Verlag, 2004.

[Sch05] Bruce Schneier. *Angewandte Kryptographie. Algorithmen, Protokolle und Sourcecode in C.* Pearson Studium, 2005.

[Sch07] Andreas Schabus. Secure code. Johannes Kepler University Linz, Secure code class, 2007.

[SL96] Vipin Samar and Charlie Lai. Making login services independent of authentication technologies. 3rd ACM conference on computer communications, 1996.

[Wae03] Dietmar Waetjen. *Kryptographie. Grundlagen, Algorithmen, Protokolle.* Spektrum Adakemischer Verlag, 2003.