On the practicality of duress passwords – An implementation for Linux authentication

Rafael Ketsetsides

Mandoulides Schools

rketsetsides@gmail.com, 00306997891864, rafket.github.io

Abstract

A duress code is a secret signal used by an individual to notify others discreetly that he is forced to do something against his will. Such codes have been used verbally, in the form of a prearranged phrase, in the military. In addition, several home alarm systems have a duress code option that can be used instead of the alarm disabling PIN. Furthermore, duress code usage has been proposed in ATM machines, again in the form of an alternative PIN code used to alert authorities. Although it is an interesting concept, it has not been examined in a cryptographically serious approach. In this paper there will be proposed several methods for implementing a password scheme with duress functionality that can be embedded easily into Linux authentication (essentially, a Pluggable Authentication Module). A duress password is the idea of duress codes applied to password authentication. Duress passwords can be categorized into *active* or *passive* passwords and *online* and *offline* passwords. The importance of duress passwords and duress codes in general lies in the fact that they may be one of the few measures that can be taken against the so called "rubber-hose" cryptanalysis method in the sense that it provides a solution to human vulnerability and even protects the human user.

Introduction

Duress codes have been proposed for use in ATM machines [3], surveillance alarm systems, [4], and mobile phone security [5]. Since their usefulness in these systems has been widely accepted, there is no reason not to adapt them to computer authentication. For this purpose, there must be a serious cryptographical approach to duress passwords in computer authentication. This paper aims to accomplish this purpose and propose some safe, working schemes.

In a normal authentication routine, a user would be prompted with a password and, upon providing it, would be granted entrance to their computer. This password may also encrypt a portion of the user's files, as is implemented in many Linux installations. There are often cases, however, that a user may be required to provide access to their account. By using a duress code, a user could grant access to their account, while secretly sending a duress signal. It is obvious

that at no part of this procedure should the adversary forcing the user be able to detect whether the password used was a duress or genuine one. For example, say Alice and Bob are undercover agents and Eve is an evil criminal. Eve has captured Bob and is forcing him to let her remote access his computer at the agency's headquarters. Bob wants to alert Alice, but under no circumstances does he want Eve to find out. Thankfully, he has a duress password scheme installed on his computer that will inform the agency that he is in trouble. Using the duress password, he will grant Eve access to his computer, but will also alert Alice. From this example, it is obvious that duress passwords often put greater value in the user's integrity than data security.

Previous work

While previous work has illustrated multiple aspects of duress codes, this paper is more focused on practicality and application in user authentication. Clark and Hengartner [@[1]] provide schemes that solve the general problem of duress codes in communication, especially in Internet-based voting. Again it could be noted that the usage of duress codes has been debated on several models that are outside the field of cryptography [3], [4], [5].

Definitions

A duress authentication scheme provides two passwords; a *main* password that the user will normally use, and a *duress* password that the user may only use in special occasions. At no point before or after authentication should there be a way to discern if a main or a duress password was used. However, due to the nature of duress passwords, it may be assumed that they will only be used once. In accordance to Kirchhoff's Principle, the adversary may have full knowledge of the scheme. The adversary may not know the following:

1. *Whether the duress scheme is used*
2. *The main and active passwords*
3. *Whether the main and duress passwords are active and if they are what action they invoke*

More formally, if

- $P_m \in \{0,1\}^*$ is defined as the main password,
- $P_d \in \{0,1\}^*$ is the duress password,
- $F_m \in \{0,1\}^*$ is the main password's action (in the form of a script or file),
- $F_d \in \{0,1\}^*$ is the duress password's action,
- $U \in \{0,1\}^*$ is the user's username,

- $S_m, S_d \in \{0,1\}^m$ are random salts,
- $Enc(Plaintext, Key) : (\{0,1\}^*, \{0,1\}^n) \mapsto \{0,1\}^*$ is a secure encryption function,
- $H(Salt, Password) : (\{0,1\}^m, \{0,1\}^*) \mapsto \{0,1\}^n$ is a key derivation function,

then $E_m = Enc(F_m, H(S_m||U, P_m))$ is the encrypted main password's action, and $E_d = Enc(F_d, H(S_d||U, P_d))$ is the encrypted duress password's action. It should be noted that the only reason U is used is to avoid confusion between user accounts with the same password (it can otherwise be considered a part of the salt), and that is why $S||U$ is considered to have the same size as S. In an implementation, a mapping function would be used on U to map it to the correct size so that it can be concatenated with the salt. Thus, from now on, U will be omitted, since it can be considered a part of the salt. E_m, E_d, U, S_m, S_d and of course $Enc()$ and $H()$ are public.

The adversary is given either P_m or P_d and his goal is to find out which one he was given. Therefore, the standard oracle's function is defined as $A(E_m, E_d, S_m, S_d, P) \mapsto \{0,1\}$ where P is either P_m or P_d, and it returns 0 if the given password is a main password and 1 otherwise. It will be proven that there is no efficient oracle $A()$. It has to be taken as granted that the adversary can deduce whether an action is a duress or a main one, if and only if he is given the action F (decrypted) and that action is non-zero. Therefore, it can be assumed that there exists an efficient oracle $C(F) \mapsto \{0,1\}, F \neq 0$ with $C(F) = 0$ when F is a main action and $C(F) = 1$ when F is a duress action.

Generally a duress scheme is secure when

$$|Pr[A(E_m, E_d, P_d) = 1] - Pr[A(E_m, E_d, X \leftarrow \{0,1\}^*) = 1]| < \epsilon$$

where ϵ is a negligible security parameter.

Active and Passive Passwords

An *active* password is defined as one that invokes an action when used. That action may be decrypting a file, alerting a third party or emptying the file-system.

A *passive* password is defined as one that does nothing when invoked, as opposed to an active password. Of course the passive password will provide normal access to the user's account (so that the scheme is not revealed). In order for a passive duress password to be of any use, the main password needs to be active. The main password may decrypt a hidden portion of the file-system, so by using the duress password the user hides some of their files. In the case of both a passive main and duress passwords, it can be assumed that the duress scheme is not in use.

Online and Offline Passwords

An *online* password is defined as a password that uses communication with another computer to call an action. For example, an online scheme could send a (prearranged) signal to a third party. If the authentication is done remotely to a computer that is not physically accessible, the duress password always acts online.

An *offline* password, as opposed to an online password, calls an action that acts only on the computer and can be considered known to the adversary. Thus, offline duress passwords must always be passive. Nevertheless, there can be an offline duress password that is offline and active (for example a password that deletes an important file), but they are not formally secure. In practice, however, these too can prove useful. Generally online duress passwords are more useful and should be chosen over offline passwords.

Based upon those definitions, the former can be expanded. A *passive* password is binded to an action F that is empty ($F = 0$), while an *active* password is binded to a non-zero action. Furthermore, the adversary cannot observe an *online* password's action (so in that case E_m and E_d are not known to him, and neither are F_m and F_d), while he can observe an *offline* password's action.

By the definition of a secure encryption function, it is known that, if $H(S, P)$ (or, equivalently P) is not given, F cannot be deduced from $Enc(F, H(S, P))$. Therefore, the adversary can only decrypt the F he is given the password to (if he is given P_m he can only decrypt E_m to get F_m and if he is given P_d he can only decrypt E_d to get F_d).

In the case of an *online* password, the existence of an oracle $A()$ is obviously impossible, because he will only be given U, S_m, S_d, and P, which contain no information about whether $P = P_m$ or $P = P_d$. On the other hand, in the case of an *offline* password, the adversary has access to E_m and E_d. However, due to the fact that the F he decrypts is zero, he cannot deduce whether he was given a passive duress or a passive main password. Therefore, in both cases, the adversary cannot use $C()$ in order to conclude whether the password used was a duress one or not.

Therefore, it can safely be concluded that, in both the cases of an online duress password and an offline passive duress password, there is no efficient standard oracle $A()$ that can predict whether a duress password was used or not.

In conclusion, in the case of an (active or passive) online scheme it has been proven that $|Pr[A(E_m, E_d, P_d) = 1] - Pr[A(E_m, E_d, X \leftarrow \{0,1\}^*) = 1]| < \epsilon$, with $P_m, P_d \in \{0,1\}^*$, $F_m, F_d \in \{0,1\}^*$, $E_m = Enc(F_m, P_m), E_d = Enc(F_d, P_d)$ and the adversary cannot deduce whether a given F is F_m or F_d. In the case of a passive offline scheme it has been proven that $|Pr[A(E_m, E_d, P_d) =$

$1] - Pr[A(E_m, E_d, X \leftarrow \{0,1\}^*) = 1]| < \epsilon$, with $P_m, P_d \in \{0,1\}^*$, $F_m \in \{0,1\}^*$, $F_d = 0$ and $E_m = Enc(F_m, P_m), E_d = Enc(F_d, P_d)$. The adversary cannot distinguish whether the function that he has decrypted is a duress or main function because he will be given a duress password, the function will be empty (harmless), and could just as well be a passive main function in an online scheme with an active duress password, or a simple main function without a duress password that just happens to call a function. Note that $F_d = 0$ does not necessarily mean that F_d does nothing, it just means that it does nothing suspicious.

It should also be noted that the scheme must provide *Plausible deniability*, that is, the user must be able to deny the use of a duress code, since that alone may be a reason to question the user's integrity. This can only be attained by using a passive duress code, since otherwise the use of the scheme is obvious. In that case, there is no way to prove that the user is using the scheme, since no other action in the database can be decrypted using the password that the user provided. Therefore, the user is able to deny the use of a duress code by him, and can claim that another user on the same system is using duress codes. Practically, the mere existence of a duress scheme in a person's computer can be enough evidence to question his integrity.

Implementation

The project's source code is maintained at github.com/rafket/pam-duress

The implementation will need to provide a way to utilize active, passive, online and offline passwords. A secure scheme will be proposed (that is mostly like normal authentication systems) that will be easily modifiable to fit the user's requirements. In this scheme each user-password combination will be assigned to an action (a script). Thus, both the main and the duress passwords can be implemented using the same code. The script is encrypted using the password as a key. The concatenated user-password combination is then salted and its hash is stored. The encryption function used (for $Enc()$) is the AES-256 block cipher in CBC mode and the key derivation function (for $H()$) is PBKDF2-SHA256 with 1000000 rounds. The salt is 16 characters long and is comprised of numbers, letters (both upper-case and lower-case) and the characters "." and "/". The salt used for the encryption is 16 characters long as well and is comprised of upper-case letters and numbers. It has been proven that PBKDF2 is secure and it is generally accepted [2]. Should the necessity arise, the system could be easily migrated to a more modern alternative, such as scrypt. The AES-256 block cipher's security has also been proved and is considered sufficient for this scheme's purpose. The library used for cryptographic functions is libOpenSSL.

In short, in this implementation, $Enc(Plaintext, Key) = AES - 256 - CBC(Plaintext, Key)$ and $H(Salt||Username, Password) = PBKDF2(SHA - 256, Username||Password, Salt, 10^6, 256)$.

The installation script creates a file that will store the hashes (in the format `salt:hash`) and a folder of scripts/executables in which the encrypted executables will be copied. The way each hash is linked to an executable is through the executable's name (the executable's name is the hash). The installation script also creates several decoy user-password entries along with scripts in order to provide a stronger argument for plausible deniability.

There is also the capability of deleting a user-password entry (along with the linked action). The deletion script hashes the given user-password combination, deletes the entry from the hash storing file and deletes the script from the scripts' folder.

Pluggable Authentication Module

The Pluggable Authentication Module, or PAM, was chosen due to its flexibility and wide use. While many criticise it, in its modern version there is no security flaw (known) and the only problem it has is the fact that it is needlessly complicated and at times slow. For the purposes of this paper, however, it is perfect since the installation of a new authentication module is extremely easy. The implementation contains a module written in C that can be incorporated into `common-auth` so that the module is used by all authentication processes. The module provided is an authentication module, and there is no password module provided in order to add a user-password combination. The library used for the PAM module is libPAM.

Randomness

Randomness is used when creating a salt. In order for randomness to be secure, OpenSSL's randomness was used, instead of the Linux randomness source `/dev/urandom`. It should be noted that OpenSSL's randomness algorithm in Linux seeds from `/dev/urandom`. Another application in the project that uses randomness is the script that creates decoy duress accounts. This script creates two random strings; one in place of the salt and one in place of the hash. It then proceeds to create a random file that appears to be encrypted. Due to the properties of PBKDF2 and AES-256 randomness is indistinguishable from the outputs of these two algorithms. The decoy script creation uses the openssl `rand` command, but the hash and salt creation and the decision upon the size of the script is done through Linux randomness (`/dev/urandom`).

Encryption

The encryption (and decryption) of the scripts is done through AES-256 CBC. The OpenSSL library is used for the implementation, so it should be considered secure. The encryption format is the same as the standard that the `openssl`

command provides. In short this format consists of a 16 bytes header (8 bytes signature and 8 bytes that describe the salt used) and the rest is the output of the AES algorithm. The signature is the string "Salted___". Since the implementation and the standard used have been reviewed numerous times, the encryption should be considered secure.

Password Hashing

The algorithm used is PBKDF2-SHA256. The number of iterations is 1000000. This number has been chosen based on estimates and choices in similar applications [6], [7]. With the fast rise in computer hardware speed and capabilities, the difficulty should be able to rise. The number of iterations is extremely easy to change, since it is set in the Makefile. It should also be noted that, although in the formal definition the username is concatenated with the salt in the $H()$ function, in the implementation the username is concatenated with the password purely for ease in implementation. This should rise no security vulnerability, since it is a way of salting. In addition, the PBKDF2 algorithm is fairly old and could be soon considered deprecated. In such a case, the scheme should migrate to a safer password-hashing model, such as the bcrypt or the scrypt algorithms.

Acknowledgements

On a personal note, the writer would like to thank Dionysis Zindros and Lazaros Tzimkas for their psychological support and review of this paper.

References

[1]

Panic Passwords: Authenticating under Duress. Jeremy Clark and Urs Hengartner

[2]

PKCS #5: Password-Based Cryptography Specification Version 2.0

[3]

Computerized password verification system and method for ATM transactions. R.K. Russiko, United States Patent, 6871288, 2005. (also cited in [1])

[4]

Intelligent surveillance alarm system and method. R.J. Massa, T.R. Ellis, R.G. LePage, United States Patent, 4589081, 1986. (also cited in [1])

[5]

Password methods and systems for use on a mobile device. A. Munje and T. Plestid, United States Patent, 11181522, 2007. (also cited in [1])

[6]

Windows Data Protection Application Programming Interface (DPAPI) in Windows XP). NAIO Labs, Network Associates, Inc. October 2001

[7]

AES Encryption Information Specification, WinZip, January 2009