A Poor Programmer's Fix for Username/Password

Michael Scott

MIRACL Labs
mike.scott@miracl.com
October 12, 2015

Abstract. You are a programmer that uses the Internet a lot. You don't want to be one of those unfortunates that gets their password hacked, due to some asshole's failure to properly protect the password file on a remote server. You don't want to change the world, you just want to work securely with the world as it is. You haven't the patience for elaborate procedures. You don't want it to cost you. How can you live with the broken Username/Password system, and yet feel certain that while the rest of the world may be vulnerable, you will be OK?

1 Introduction

First let us quickly recall how Username/Passsword works. The only cryptographic tool required is the humble hash function like the standard SHA256, which converts a string into a unique 256-bit fingerprint. So as a function it works like $y = H(x)$ where x is a string, and y is the fingerprint. Inputs can be concatenated, so $x = H(a, b)$ just means that the string b is appended to the end of a to provide the input.

Once you register, the server generates a random "salt" s and stores $s, H(p, s)$ against your username, where p is your password. All this information for every service user or client is stored in a password file. When you enter your username and password, the server looks up your salt, calculates $H(p, s)$ and compares it with the stored value. If its good, you are in. Which is all very fine and dandy, but then a hacker steals the password file, and using a dictionary of common passwords, tries every one until they start getting hits. Your only protection is to make sure and use a password which is not in that dictionary, hence those tedious password policies requiring you to remember an awful looking password for every service you use, and which you really need to write down somewhere. For the hacker who has stolen the password file they are not necessarily motivated to hack you individually. In fact they probably know nothing about you in advance of the attack. For them, its not personal.

2 Attacks and Defenses

First let us step back a bit and look at the issue from the hacker point of view. For a hacker the aim is often to maximimize the bang for the buck. In other words with the minimum amount of time and effort they would like to capture as many individual credentials as possible. Attacks can be classified as scalable or non-scalable. A scalable attack captures the credentials of multiple individuals with just one attack. A scalable attack might be launched against the server as described above. Or it may be launched by spreading a virus that attacks infected apps on the client side, capturing the Username/Password combination and reporting them back to the hacker. A non-scalar attack on the other hand requires an individualized effort again each individual target. Non-scalar attacks therefore tend to be targeted, and if for some reason you have personally attracted the attention of a talented hacker, particularly if they work for a well resourced government agency, you are probably in big trouble.

One major weakness of the Username/Password mechanism is that the Username and the Password are transmitted directly to the server, albeit under cover of the SSL protocol. But this does imply that at both ends of the link there are potential points of attack, maybe by a virus, or by an employee on the server side. A much better idea would be a challenge-response based protocol in which the client secret is never transmitted from client to server, but rather used to construct a specific response to a random server challenge.

But for this exercise we will simply try and improve Username/Password as much as we can. Our plan is to avoid scalable attacks (a) on the server side by using very strong passwords, and (b) on the client side by deploying an individualized app, requiring a specific and targeted non-scalable attack to hack. We believe that just by forcing a hacker to go after individuals, rather than harvesting millions of passwords in a single attack, that this is enough to improve the user's security by many degrees of magnitude.

For more complete protection there really is a need for a better two-factor zero-knowledge challenge-response protocol, like our own M-Pin. But that's another story.

3 The Poor Programmer's Protocol

Consider now a specially tailored password generator app. Stored in the source code somewhere is a random string, created by the programmer getting their 3 year old grandson to bang on the keyboard

```
char r[]="dfgneNVriothjOerjgorigjq[89u8qu1j)(kdUjSSgj4[jjjr";
```

The program consists of a drop-down list with the names of the services with which the programmer has an account. There is also a box into which they enter their one-and-only Master Password p when the program first starts up, and another box which requires a short fixed 4-digit PIN number. When they select the name of a service n from the drop-down list, a suitable password for the selected service is calculated as $pw_n = H(p, H(PIN, H(r, H(n))))$, and automatically copied to the clipboard. In other words the password is generated from a mix of the Master Password, the PIN, the name of the service (like "GitHub"), and the random string.

The user only needs to enter the master password once when they activate the app at the start of their working day. When they navigate to a service on the internet, they click on the app, enter the fixed PIN, and click on the name of the service from the drop-down list, and finally click "paste" in the service's password field. And thats it. Easier than typing in a long complex password. Note that the entered password pw_n is never visible, and so remains unknown even to the user. We are assuming of course that the average user has no problem remembering one reasonably secure password and a PIN number.

The binary of the app can be copied to other computers and devices that the programmer owns. Its source code can be kept offline on a USB key.

4 Security

One of the biggest disincentives for a hacker is to have to go after each service user individually, and hack into their individual device. That is a major hassle, and is probably outside of their skill set. And one component of the password is the random r tucked inside the binary of your very own app, a copy of which is not available to anyone else on the planet. And you can get creative with r, and where it is stored in the source code. Although the original source code might be open source (see below), as a programmer you can modify it. For example you can spread pieces of r all over the program, and recombine them programmatically. Just do enough to make your binary unique.

Ideally your one-and-only Master Password should not be in any dictionary, certainly when combined with the PIN. One attack is to steal the password file from a server, then access your computer and try password and PIN guesses until they get a hit on the password file entry. A standard way to make password guessing more difficult would be to calculate it as $pw_n = H(H(H(H(H(H(H....(H(p)))....)))), H(PIN, H(r, H(n)))$, with p hashed maybe thousands of times, so that it takes a few seconds to calculate. Indeed since you only enter it once during the working day, you might afford to wait even a minute or two.

Of course if your app binary is stolen it can be reverse engineered and r extracted. But first they got to get your app binary, and perform a specially tailored attack on it, because there is none other quite like it. If some-one sits down at your computer while the app is active, they cannot generate your passwords, as they don't know the PIN. Note that nothing related to the Master Password or the PIN is stored inside of the app, so reverse engineering your app doesn't compromise either.

Appendix

OK, lets do it. First download the programming tools from `www.qt.io`. These allow you to develop the application across a range of devices and operating systems. I just did it for a Windows desktop PC. Pay attention to the licensing arrangements, although if I read it right, and since its for personal use, its free.

The program consists of just 4 files.

- main.qml – This describes the extremely simple user interface
- qpass.h – a header file
- main.cpp – the main C++ file
- hash.cpp – the hashing code and some utility code

As always when implementing, some issues arise. To create the passwords we basically convert the output of the final hash to base64 encoding, which gives us a mix of upper and lower case letters and digits, and occasional special symbols as dictated by most password policies. However we check the output to ensure it contains at least one uppercase letter, one lowercase letter and one digit. If it doesn't we hash it again until it does. Another decision is the length of the password. We decided on 12 characters, giving 72-bits of entropy. So a typical password generated by this program looks like `hSUgpKA09v6Z`.

Here is the main.qml file. Observe where you enter your list of services. If you want to add another one later, pop it in here and recompile the app.

```
import QtQuick 2.3
import QtQuick.Controls 1.2

ApplicationWindow {
    visible: true
    width: 280
    height: 150
    maximumHeight: 150
    maximumWidth: 280
    title: qsTr("Password Manager")

    TextInput {
        x: 100
        y: 20
        width: 160
        focus: true
        enabled: true
        objectName: "favorite"
    }

    TextField {
        x:100
        y:60
        width:50
        echoMode: TextInput.Password
        maximumLength: 6
        enabled: false
        objectName: "pin"
    }

    ComboBox {
        x: 100
        y: 100
        width: 160

 // *** Enter your own list of services here

        model: [ "None", "GitHub", "Stash", "Facebook" ]
        objectName: "service"
        enabled: false
    }

    Label {
        x: 10
        y: 20
        text: qsTr("Master Password")
    }

    Label {
        x: 10
        y: 60
        text: qsTr("Enter PIN")
    }

    Label {
        x: 10
        y: 100
        text: qsTr("Choose Service")
    }
}
```

Next the header file qpass.h.

```
#include <QObject>

#ifndef QPASS
#define QPASS

class QPass : public QObject
{
    Q_OBJECT

private:
    QString pwdtext;
    QString srvtext;
    int PIN;
    QObject *pwd,*srv,*pin;
    char ph[32],digest[32];
public:
    QPass(QObject *pwdi,QObject *srvi,QObject *pini) { pwd=pwdi; srv=srvi; pin=pini; }

public slots:
    void setpwValue();
    void setsvValue();
    void setpnValue();
};

#define unsign32 unsigned int

typedef struct {
unsign32 length[2];  /**< 64-bit input length */
unsign32 h[8];       /**< Internal state */
unsign32 w[64]; /**< Internal state */
} hash;

extern void HASH_init(hash *H);
extern void HASH_process(hash *H,int b);
extern void HASH_hash(hash *H,char *h);

extern void tobase64(char *,char *);
extern void HASH_again(char *);
extern int policy(char *);

#define PASSLEN 9 // password lengh in bytes - multiple of 3
#define HCOUNT 10000 // hash iterations
```

```
#endif // QPASS
```

Now the main C++ program. Observe the place where you (or your grandson) enters the long random string.

```
#include <QApplication>
#include <QQmlApplicationEngine>
#include <QClipboard>
#include <String>
#include "qpass.h"

// *** Enter your own random number here

QString r="jijvijHJodP67odld0osjJhHuU7Yshsh&jssKskJS5sgSHLSl;sSss";

// Master password entered
void QPass::setpwValue()
{
    hash sh;
    QByteArray ba;

    pwdtext=pwd->property("text").toString();
    pwd->setProperty("text","");

    // First hash Master Password

    HASH_init(&sh);
    ba=pwdtext.toLatin1();
    for (int i=0;i<ba.length();i++)
        HASH_process(&sh,ba[i]);
    HASH_hash(&sh,ph);

    pwdtext.clear();
    // digest=H(H(H(H(pw)))))
    for (int i=0;i<HCOUNT;i++) HASH_again(ph);

    pin->setProperty("enabled",true);
    pin->setProperty("focus",true);
    pwd->setProperty("enabled",false);

}

// PIN entered
void QPass::setpnValue()
{
    hash sh;
    PIN=pin->property("text").toInt();
    srv->setProperty("enabled",true);
    pin->setProperty("text","");
    pin->setProperty("enabled",false);

    HASH_init(&sh);
    HASH_process(&sh,PIN%100);
    HASH_process(&sh,(PIN/100)%100);
    HASH_process(&sh,PIN/10000);
    for (int i=0;i<32;i++) HASH_process(&sh,ph[i]);
    HASH_hash(&sh,digest);
}

// Service picked from list
void QPass::setsvValue()
{
    QChar ch;
    hash sh;
    char b64[50];
    QByteArray ba;

    if (srv->property("currentIndex")==0) return;

    srvtext=srv->property("currentText").toString();

    HASH_init(&sh);
    ba=srvtext.toLatin1();
    for (int i=0;i<ba.length();i++)
        HASH_process(&sh,ba[i]);
    for (int i=0;i<32;i++) HASH_process(&sh,digest[i]);
    HASH_hash(&sh,digest);

    //digest=H(Service,H(H(H(H(pw)))))

    HASH_init(&sh);
    ba=r.toLatin1();
    for (int i=0;i<ba.length();i++)
        HASH_process(&sh,ba[i]);
    for (int i=0;i<32;i++) HASH_process(&sh,digest[i]);
    HASH_hash(&sh,digest);

    //digest=H(r,H(Service,H(H(H(H(pw))))))

    tobase64(b64,digest);  // Convert to base64

    // check it meets policy requirements - else hash again until it does
    while (!policy(b64))
    {
        HASH_again(digest);
        tobase64(b64,digest);
    }

    // push password onto Clipboard
    QClipboard *clipboard = QApplication::clipboard();
    clipboard->setText(b64);

    srv->setProperty("currentIndex",0);
    srv->setProperty("enabled",false);
    pin->setProperty("enabled",true);
    PIN=0;
    for (int i=0;i<32;i++) digest[i]=0;
}

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QQmlApplicationEngine engine;
    engine.load(QUrl(QStringLiteral("qrc:/main.qml")));

    // Access the root object
    QObject *rootObject = engine.rootObjects().first();

    // Access the elements
    QObject *Favorite = rootObject->findChild<QObject*>("favorite");
    QObject *Service = rootObject->findChild<QObject*>("service");
    QObject *Pin = rootObject->findChild<QObject*>("pin");

    QPass Password(Favorite,Service,Pin);

    // Set up the signals and slots
    engine.connect(Service,SIGNAL(currentIndexChanged()),&Password,SLOT(setsvValue()));
    engine.connect(Favorite,SIGNAL(accepted()),&Password, SLOT(setpwValue()));
    engine.connect(Pin,SIGNAL(accepted()),&Password, SLOT(setpnValue()));

    // let rip
    return app.exec();
```

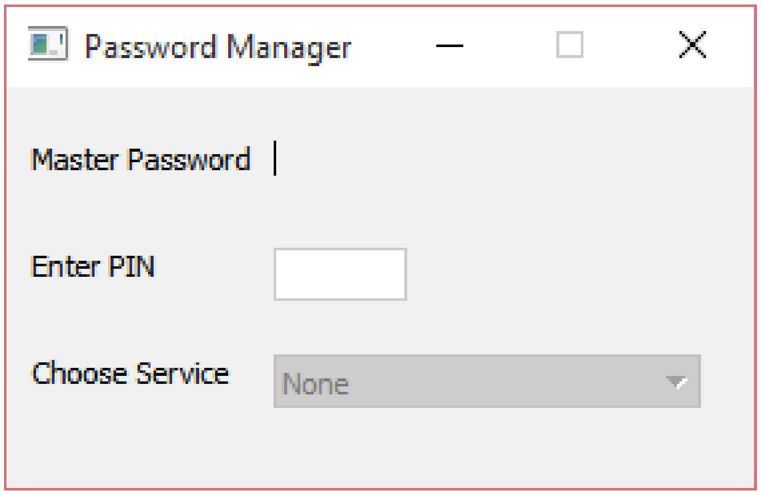
```
}
```

Finally the file hash.cpp which contains the SHA256 hashing code and some utility functions.

```
/*
 * Implementation of the Secure Hashing Algorithm (SHA-256)
 *
 * Generates a 256 bit message digest. It should be impossible to come
 * come up with two messages that hash to the same value ("collision free").
 *
 * For use with byte-oriented messages only. Could/Should be speeded
 * up by unwinding loops in HASH_transform(), and assembly patches.
 */

#include <cctype>
#include "qpass.h"

#define H0 0x6A09E667L
#define H1 0xBB67AE85L
#define H2 0x3C6EF372L
#define H3 0xA54FF53AL
#define H4 0x510E527FL
#define H5 0x9B05688CL
#define H6 0x1F83D9ABL
#define H7 0x5BE0CD19L

static const unsign32 K[64]={
0x428a2f98L,0x71374491L,0xb5c0fbcfL,0xe9b5dba5L,0x3956c25bL,0x59f111f1L,0x923f82a4L,0xab1c5ed5L,
0xd807aa98L,0x12835b01L,0x243185beL,0x550c7dc3L,0x72be5d74L,0x80deb1feL,0x9bdc06a7L,0xc19bf174L,
0xe49b69c1L,0xefbe4786L,0x0fc19dc6L,0x240ca1ccL,0x2de92c6fL,0x4a7484aaL,0x5cb0a9dcL,0x76f988daL,
0x983e5152L,0xa831c66dL,0xb00327c8L,0xbf597fc7L,0xc6e00bf3L,0xd5a79147L,0x06ca6351L,0x14292967L,
0x27b70a85L,0x2e1b2138L,0x4d2c6dfcL,0x53380d13L,0x650a7354L,0x766a0abbL,0x81c2c92eL,0x92722c85L,
0xa2bfe8a1L,0xa81a664bL,0xc24b8b70L,0xc76c51a3L,0xd192e819L,0xd6990624L,0xf40e3585L,0x106aa070L,
0x19a4c116L,0x1e376c08L,0x2748774cL,0x34b0bcb5L,0x391c0cb3L,0x4ed8aa4aL,0x5b9cca4fL,0x682e6ff3L,
0x748f82eeL,0x78a5636fL,0x84c87814L,0x8cc70208L,0x90befffaL,0xa4506cebL,0xbef9a3f7L,0xc67178f2L};

#define PAD  0x80
#define ZERO 0

/* functions */

#define S(n,x) (((x)>>n) | ((x)<<(32-n)))
#define R(n,x) ((x)>>n)

#define Ch(x,y,z)  ((x&y)^(~(x)&z))
#define Maj(x,y,z) ((x&y)^(x&z)^(y&z))
#define Sig0(x)    (S(2,x)^S(13,x)^S(22,x))
#define Sig1(x)    (S(6,x)^S(11,x)^S(25,x))
#define theta0(x)  (S(7,x)^S(18,x)^R(3,x))
#define theta1(x)  (S(17,x)^S(19,x)^R(10,x))

/* SU= 72 */
static void HASH_transform(hash *sh)
{ /* basic transformation step */
    unsign32 a,b,c,d,e,f,g,h,t1,t2;
    int j;
    for (j=16;j<64;j++)
        sh->w[j]=theta1(sh->w[j-2])+sh->w[j-7]+theta0(sh->w[j-15])+sh->w[j-16];

    a=sh->h[0]; b=sh->h[1]; c=sh->h[2]; d=sh->h[3];
    e=sh->h[4]; f=sh->h[5]; g=sh->h[6]; h=sh->h[7];

    for (j=0;j<64;j++)
    { /* 64 times - mush it up */
```

```
        t1=h+Sig1(e)+Ch(e,f,g)+K[j]+sh->w[j];
        t2=Sig0(a)+Maj(a,b,c);
        h=g; g=f; f=e;
        e=d+t1;
        d=c;
        c=b;
        b=a;
        a=t1+t2;
    }

    sh->h[0]+=a; sh->h[1]+=b; sh->h[2]+=c; sh->h[3]+=d;
    sh->h[4]+=e; sh->h[5]+=f; sh->h[6]+=g; sh->h[7]+=h;
}

/* Initialise Hash function */
void HASH_init(hash *sh)
{ /* re-initialise */
    int i;
    for (i=0;i<64;i++) sh->w[i]=0L;
    sh->length[0]=sh->length[1]=0L;
    sh->h[0]=H0;
    sh->h[1]=H1;
    sh->h[2]=H2;
    sh->h[3]=H3;
    sh->h[4]=H4;
    sh->h[5]=H5;
    sh->h[6]=H6;
    sh->h[7]=H7;
}

/* process a single byte */
void HASH_process(hash *sh,int byte)
{ /* process the next message byte */
    int cnt;
//printf("byt= %x\n",byte);
    cnt=(int)((sh->length[0]/32)%16);

    sh->w[cnt]<<=8;
    sh->w[cnt]|=(unsign32)(byte&0xFF);

    sh->length[0]+=8;
    if (sh->length[0]==0L) { sh->length[1]++; sh->length[0]=0L; }
    if ((sh->length[0]%512)==0) HASH_transform(sh);
}

/* SU= 24 */
/* Generate 32-byte Hash */
void HASH_hash(hash *sh,char digest[32])
{ /* pad message and finish - supply digest */
    int i;
    unsign32 len0,len1;
    len0=sh->length[0];
    len1=sh->length[1];
    HASH_process(sh,PAD);
    while ((sh->length[0]%512)!=448) HASH_process(sh,ZERO);
    sh->w[14]=len1;
    sh->w[15]=len0;
    HASH_transform(sh);
    for (i=0;i<32;i++)
    { /* convert to bytes */
        digest[i]=(char)((sh->h[i/4]>>(8*(3-i%4))) & 0xffL);
    }
    HASH_init(sh);
}

void HASH_again(char *d)
{
    int i;
    hash sh;
    HASH_init(&sh);
    for (i=0;i<32;i++) HASH_process(&sh,d[i]);
    HASH_hash(&sh,d);
}

/* Convert byte array to base64 string */
void tobase64(char *b,char *w)
{
    int i,j,k,rem,last;
    int c,ch[4],len=PASSLEN;
    unsigned char ptr[3];
    rem=len%3; j=k=0; last=4;
    while (j<len)
    {
        for (i=0;i<3;i++)
        {
            if (j<len) ptr[i]=w[j++];
            else {ptr[i]=0; last--;}
        }
        ch[0]=(ptr[0]>>2)&0x3f;
        ch[1]=((ptr[0]<<4)|(ptr[1]>>4))&0x3f;
        ch[2]=((ptr[1]<<2)|(ptr[2]>>6))&0x3f;
        ch[3]=ptr[2]&0x3f;
        for (i=0;i<last;i++)
        {
            c=ch[i];
            if (c<26) c+=65;
            if (c>=26 && c<52) c+=71;
            if (c>=52 && c<62) c-=4;
            if (c==62) c='+';
            if (c==63) c='/';
            b[k++]=c;
        }
    }
    if (rem>0) for (i=rem;i<3;i++) b[k++]='=';
    b[k]='\0';
}

/* return 1 if password meets password policy requirement, else 0 */
int policy(char *b)
{
    int i,isd,isl,isu;
    isd=isl=isu=0;
    for (i=0;i<PASSLEN;i++)
    {
       if (isdigit(b[i])) isd=1;
       if (islower(b[i])) isl=1;
       if (isupper(b[i])) isu=1;
    }
    if (isd && isl && isu) return 1;
    return 0;
}
```

When the app is running it looks like this. Not very impressive looking. The Master Password has been entered, and then disappears. All the user has to do is enter the PIN, select a service from the drop-down list and then paste the automatically calculated password into the service's password field.



Password Manager

Master Password

Enter PIN

Choose Service: None

Fig. 1. The Password Manager app.