YubiKey Static Password Function

White Paper

November 11, 2015

By Dain Nilsson

Copyright

Trademarks

Disclaimer

Contact Information

Yubico Inc
420 Florence Street, Suite 200
Palo Alto, CA 94301
USA
yubi.co/contact

Contents

Introduction

The YubiKey has evolved over the years. As hardware improves, new functions become possible. If we go back in time and look at the initial YubiKey, the single function was YubiKey one-time password (OTP). Today, we have state-of-the-art authentication protocols like Universal 2nd Factor (U2F), asymmetric encryption using OpenPGP and PIV, and so on.

Often when Yubico makes a decision to add new functionality, we know who the target user is going to be and what impact the feature will have. A long-time function of the YubiKey that proved to have a much larger audience than anticipated, is the static password. This functionality, as you may have already guessed from the title, is also the topic of this white paper.

How It All Began

In the early days of the YubiKey, someone had a pretty simple idea. We had this device that acts as a USB keyboard and is capable of “typing” unique OTPs triggered by a button touch. What if you could program it with a normal password and use it with any existing application? On one hand, a static password is not nearly as safe as an OTP. On the other hand, the number of existing systems that already support it is enormous. In a sense, you’d have a super simple password manager, requiring nothing but a USB port. Simple enough in concept, but there are several pitfalls in implementing this. This is why the functionality initially evolved over time, transforming a bit with each iteration.

Password Minus the One-Time

Yubico’s first go at this was to take the existing OTP functionality and strip out the “one-time” part. That meant freezing the internal counters, timer values, and so on, causing the same algorithm that generates a new OTP each time to generate a single password that doesn’t change. This approach had some problems.

First off, the password can’t be stored directly since it’s the result of several parameters that are programmed into the YubiKey. If these parameters are lost, you can’t program another YubiKey with the same password. A second issue is that the generated passwords only use the ModHex alphabet (explained later), which uses 16 different lowercase characters. While this is a perfectly safe password to use (more on that later), many applications mandate the inclusion of specific classes of characters, including mixed case, numbers, symbols, and so on. Even worse, many applications limit the size of passwords in length, disallowing the static OTPs due to their length of 32 (or more) characters. We worked around some of these issues by allowing shorter passwords, and doing some basic transformations to the output to ensure there is at least one symbol, one number, and one uppercase character. The issue to freely set your own password remained, and thus we added another mode of operation: scan code mode.

Primer: USB Keyboards

Here is how keyboards work. When the user presses a key, that key has a code it sends to the computer. The computer uses a table to look up that code and determine which character to print to the screen. A user's computer is configurable for a specific keyboard layout and its associated table. An example: When I type the *'Y'* key (without pressing shift) on my keyboard, it gets translated into the scan code `0x1c`*.* The operating system uses my computer's keyboard layout (Swedish) to convert that code back into the character `y`. Because the `Y` key is in the same position on a US keyboard, the scan code is interpreted as a `y` if my keyboard layout had been set to US If I had been using a German keyboard layout, however, that code is interpreted as a `z`, since German keyboards have the letter `Z` in that key position. Why am I telling you all this? Because this will help you understand how the scan code mode works with the YubiKey, and its specific issues.

YubiKey Static Password - Scan Code Mode

Now, back to static passwords on the YubiKey. To allow storage of a user provided password on a YubiKey, we introduced the scan code mode. In this mode, the user provided a list of scan codes, and the YubiKey simply presented those codes, in order. No more freezing counter values or performing crypto functions, just plain data. This let the user configure a password by choosing any of the keys available on a keyboard, including using the `Shift` key in combination with these keys.

A limitation of the YubiKey, however, prevents you from choosing characters that require a modifier key other than `Shift`. An example is the @ symbol when using a Swedish layout, which requires you to press the `Alt Gr` key in combination with the 2 key. Now, what happens if you program a password using scan codes for a US keyboard layout, and use it in a computer configured to use a German keyboard layout? As you might suspect, the password that shows up on your screen is different from the one you expected. The reason, as explained in the previous section, is that the scan code stored on the YubiKey just identifies the location of the key on a keyboard. The interpretation of that key depends on the keyboard layout configured on the computer.

Another issue centers on how the user is supposed to know that the scan code data for the password `Password`, on a US keyboard layout, is `930416161a121507`? Well, we'll try to help you out there by providing scan code tables for common keyboard layouts in our YubiKey Personalization Tool. If your language is missing, and you're feeling a bit adventurous, you can try changing the layout of your computer to US English before programming the YubiKey with your password. Then begin by pressing the keys of your keyboard like you would as usual and ignore that the output isn't right. The scan codes will be correct for your layout, and once you switch back to that layout on your computer, the output of your YubiKey should be correct as well.

ModHex to the Rescue

So it seems we can use the scan code mode for static passwords, if we're willing to jump through a few hoops. But if keyboards use scan codes, and YubiKey OTPs are created by a "keyboard," doesn't that mean that YubiKey OTPs should have this same issue when used with computers with different keyboard layouts? Indeed it does, and that is the reason YubiKey OTPs use the ModHex alphabet in the first place. The 16 characters used in the ModHex alphabet are: `c,b,d,e,f,g,h,I,j,k,l,n,r,t,u,v`. These characters share a property that makes them very valuable to a YubiKey: They use the same scan codes across a very large number of keyboard layouts. In other words, the scan code `0x06` maps to the character c for English, Swedish, German, French, and many others. Using only ModHex characters makes it easy to map characters into scan codes when programming the password, and gives you a password that will work between many different keyboard layouts.

Because of this characteristic, we recommend the use of ModHex characters for your passwords. You can use uppercase ModHex characters if needed, and include a number and a symbol if the service you're using requires it for your password. This ensures that you don't have to deal with many of the problems associated with multiple keyboard layouts.

Why Does My Password Look So Weak?

If you're using a ModHex-only password, it might look similar to this: `ihnhihgdgunndfggejdignlncejrhegb`. At first glance, that might not look very random. Can we really trust it to be secure? The reason it looks that way is because it's in ModHex, and it actually encodes quite a bit of "randomness." As previously explained, there are 16 different characters available in the ModHex alphabet. That's a lot fewer than the number of characters typically available to us, which includes everything from uppercase and lowercase to numbers and symbols. The thing that makes a ModHex-only password strong is the length, and the fact that all characters are chosen at random.

The random part defeats the most common password cracking attack, which is known as a dictionary attack. Since a lot of people choose actual words as their passwords, it's much easier to guess these passwords by first testing it against all the words in a dictionary. I've never seen `ihnhihgdgunndfggejdignlncejrhegb` in any dictionary, so an attacker would get little help from a dictionary. Instead, the attacker has to try out all possible passwords in the ModHex space to be sure to guess mine. Or try half of them to have a 50% chance of getting it, and so on. So, how many variations of this seemingly simple strings of characters are there? Let's do the math!

A ModHex password is 32 characters, each character 1 of 16 possible values. That means the total number of possible passwords using this scheme is:

16^32 or 340,282,366,920,938,463,463,374,607,431,768,211,456

That's a lot of passwords. Let's assume an attacker is capable of making one trillion (1,000,000,000,000) password guesses per second. In this equation, it would take the attacker 340,282,366,920,938,463,463,374,607 seconds to try out all combinations, or 10,790,283,070,806,014,188 years. If the attacker started brute-forcing the password now, by the time the sun burns out in five billion years they will have approximately a 1 in 2,158,056,614 chance of having guessed the right one.

How Long is Long Enough?

YubiKey has a static password setting that allows you to have a 16-character ModHex password instead of 32 characters, which obviously lowers the security by a large amount. In fact, assuming the same capability of one trillion guesses per second, it takes only a little over half a year to guess all possible passwords. Is this 16-character ModHex password useless? Well, you also have to consider context and real-world implications. If you're authenticating over the internet, the server should be blocking any brute-force attempts based on sheer volume. Even if the system does no such checks, just the fact that authentication is done over the internet severely limits the number of attempts. Assuming an attacker can somehow get away with attempting one million (1,000,000) guesses per second without raising any suspicion on the server, it would then take over half a million years to crack it. In other words, while a 16-character ModHex password may not be safe for everything (like using it to encrypt confidential files), there are still many valid use cases for it.

Final Thoughts

Scan codes or static OTPs, long or short, ModHex or not, we can’t forget that these are all just passwords and should be avoided as the only factor for authentication. We’ve talked about guessing passwords through brute-force, but there are other attacks against passwords that work regardless of length. You can be tricked into entering your password into a bogus website. Sites can get hacked and leak your password. Unencrypted transmissions can be intercepted. For these reasons and many more like them, we will always recommend the use of two-factor authentication such as U2F, wherever possible. We’re doing our part to spread the word, are ***you***?