info@thinkst.com
research@thinkst.com
http://www.thinkst.com

On Screen Keyboards Considered Harmful Today

(If the 90's are coming back, we are bringing back "Shoulder Surfing")

2011 - haroon@thinkst.com

On Screen Keyboards Considered Harmful Today

TL;DR

The On-Screen-Keyboards used on modern mobile devices (phones & tablets) leave us more vulnerable to shoulder surfing attacks. (Attacks can be trivially automated and carried out over great distances). We mask passwords on screen to prevent these attacks, but the modern devices take us 2 steps backwards in this regard. We demonstrate code we have written to automate these attacks and show the resultant attacks in action [http://thinkst.com/ocv/vids/]

Shoulder Surfing

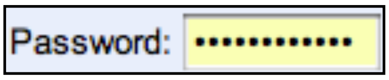
28 minutes into the movie sneakers, we see Robert Redford's team trying to use video surveillance to grab their targets password.

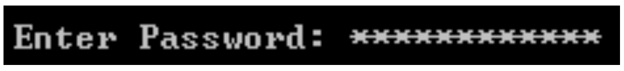
Shoulder Surfing in "Sneakers"

Bishop:	This is good. He's gonna type in his password, and we're gonna get a clear shot.
	...
Mother:	I'll blow it up.
	[Video Plays]
Bishop:	There.
Mother:	"W"
Bishop:	"G"
Carl:	Looks like the "H".
Mother:	Where do you get "H"?
Carl:	Next to the "L".
Bishop:	Come on. Do it again.
Mother:	"H" isn't next to the "L".
	[Video Plays]
Mother:	Okay, that's definitely "W", "G".
Crease:	Definitely not. No, that's a "V".

Clearly snooping passwords through surveillance and "shoulder surfing" is prone to error.

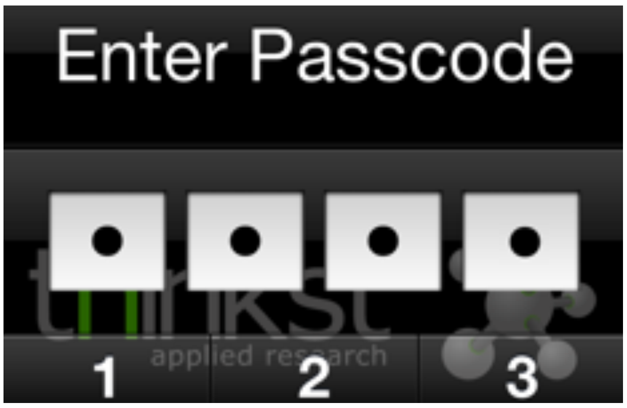
We have long realised the danger of having passwords stolen through shoulder surfing attacks which is why it is truly rare to find an application that fails to mask the password on screen.

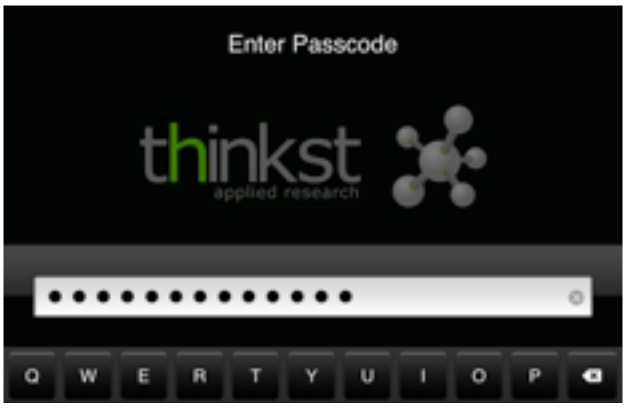




Typical Password Masking

Even the iDevices (which we examine below) mask passwords by default.



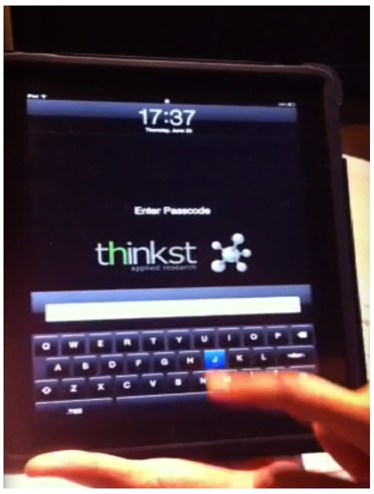


iPhone and iPad Password Entry

We take the fact that password masking is so ubiquitous as the obvious acknowledgement of shoulder surfing as a viable attack method.

Interestingly enough, in order to provide feedback to the user, most modern devices will mask the entered password, while simultaneously highlighting the keys being pressed.

(If the mathematician being spied on in "Sneakers" was using an iPad, they would have had his password easily!)



Masking Fail ?

Automating the Attack

We decided to build a simple tool to demonstrate how this attack could be automated (because we really wanted an excuse to play with OpenCV).

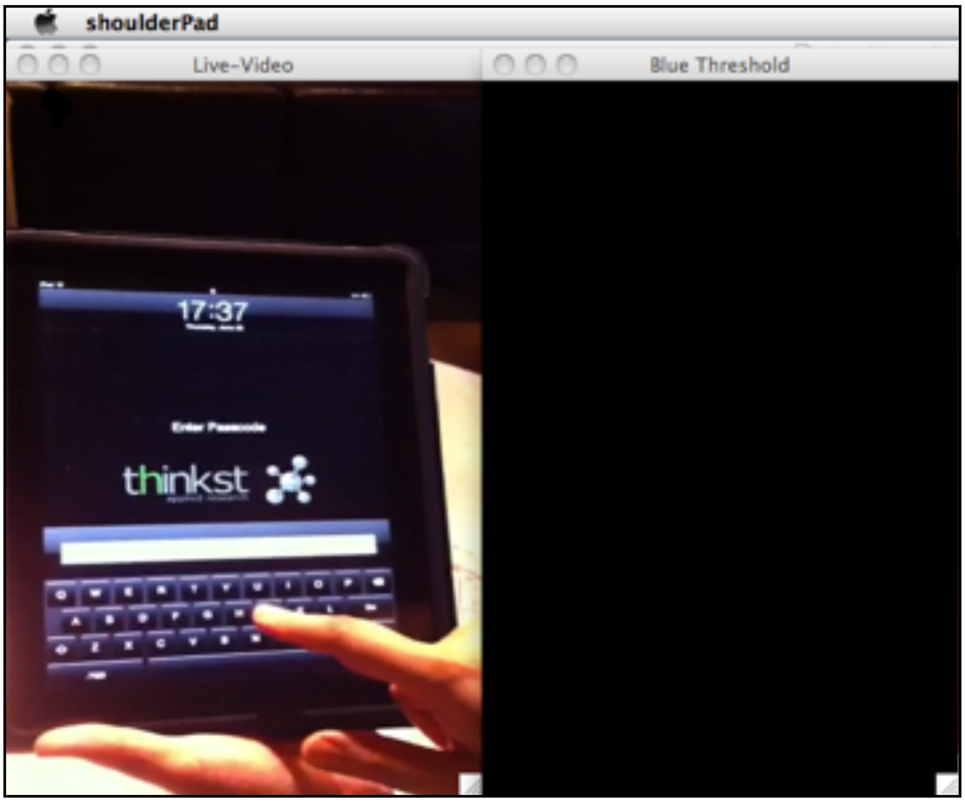
OpenCV (Open Source Computer Vision Library) is the awesome BSD licensed library (developed by Intel) aimed at real time computer vision.

It allows people like me (who previously didn't know the difference between color, chroma, hue or saturation) to build image recognition utilities. Since OpenCV compiles on the iPhone, it means that we can build a simple iPhone app, that can be aimed at a victim using an iPad to obtain his password.

(The app is built to work on captured video, so it's source can come directly from the camera, or from captured surveillance footage ala "Sneakers").

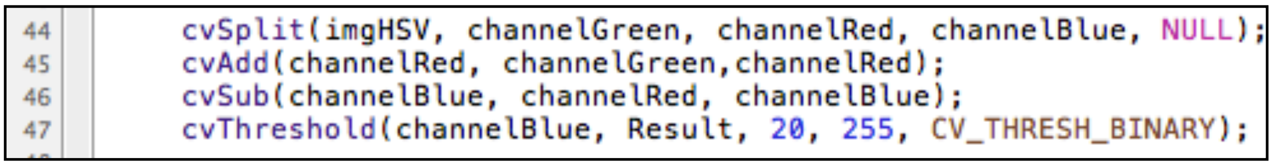
How it's Done:

We open the video stream, and iterate through captured frames one at a time using the standard openCV functions. We then duplicate the captured frame for manipulation.



Duplicating the Frame

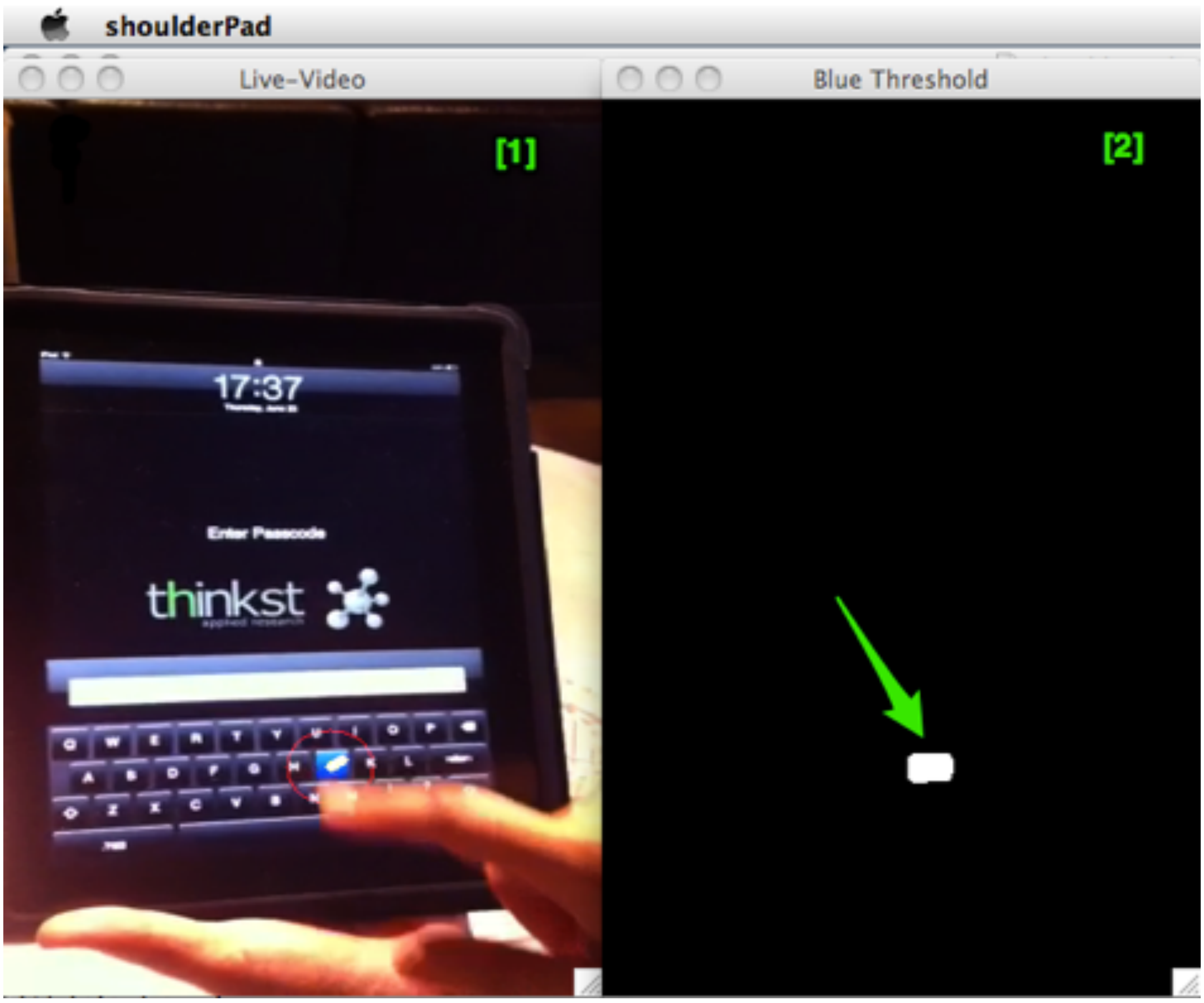
Since the iPad highlights keys in blue when clicked, we split the duplicate image into its distinct RED, BLUE and GREEN channels. We then set a threshold, effectively viewing only the blue objects in the image.



```
    cvSplit(imgHSV, channelGreen, channelRed, channelBlue, NULL);
    cvAdd(channelRed, channelGreen,channelRed);
    cvSub(channelBlue, channelRed, channelBlue);
    cvThreshold(channelBlue, Result, 20, 255, CV_THRESH_BINARY);
```

Blue Thresholding

This makes key-presses more pronounced (as seen in window [2] below).



Blue Thresholding to Detect Button Presses

Grabbing the location of the Blue blob in image [2] allows us to obtain a strong X/Y coordinate for the pressed key, which allows us to (automatically) mark the location of the key-press with a circle (in image [1]).

We now need to convert our captured X/Y coordinate to an actual keyboard key.

A Small Challenge

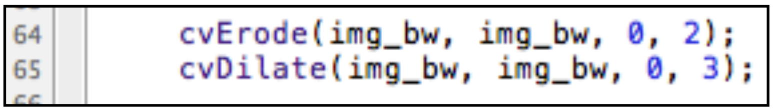
A simple mapping has 3 obvious failings:

1. **Movement**: The application determines the X/Y coordinate of the pressed key at a point in time. Since the iPad is moving (in the users hand), the same coordinate could be referring to 2 different keys a few seconds apart.
2. **Scale of Image**: The cameras distance from the screen affects the size of the keys, preventing simple solutions.
3. **Orientation**: The image might might not be straight (or could alter it's orientation slightly) during the capture.

It would obviously be possible to correct these issues by editing the captured video, but we wanted a more elegant solution. Such a solution presents itself by using the password entry box for image orientation.

For purposes of demonstration, we make 2 more copies of each frame.

The intention here was to threshold for white, the same way we previously thresholded for blue. Due entirely (i'm sure) to my lack of understanding of color and thresholding, this turned out to be harder than I previously thought. Instead we simply convert one image to black and white, and use openCV functions to dilate and erode the image a little. (What we are doing here is trying to remove background noise).



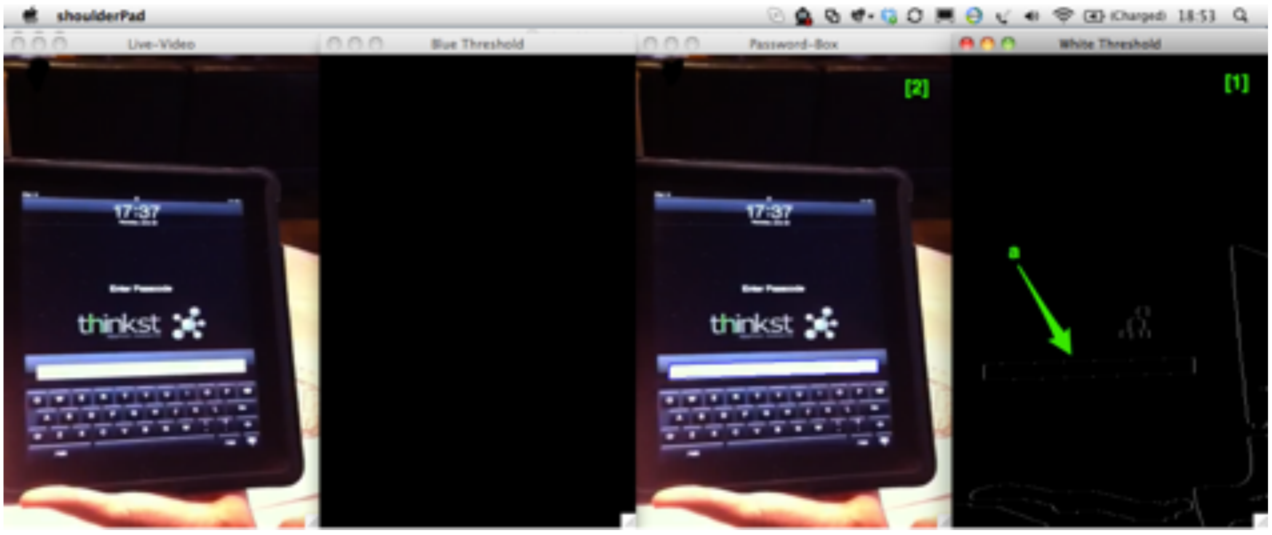
```
    cvErode(img_bw, img_bw, 0, 2);
    cvDilate(img_bw, img_bw, 0, 3);
```

OpenCV Image Clean Up

We then use the openCV version of the Canny edge detection algorithm [http://en.wikipedia.org/wiki/Canny_edge_detector] to find all of the squares in the thresholded image. (Displayed as [1] below). Depending on the background and other objects in the vicinity (despite our erode/dilute dance), we could still find a number of edges in the image. To avoid this, we do 2 more checks:

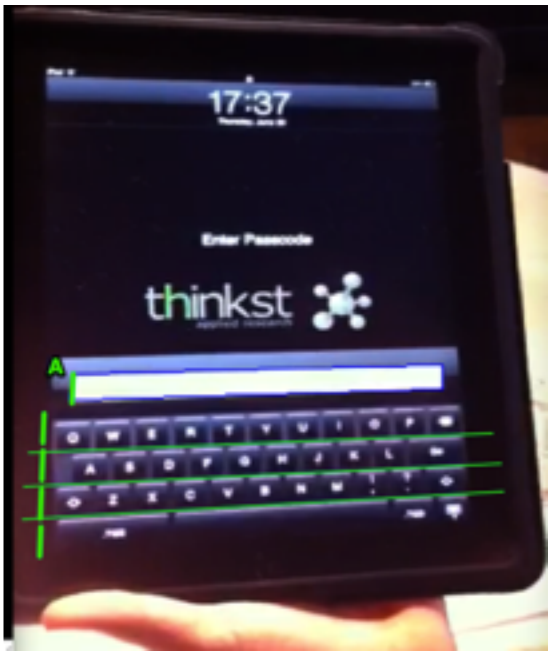
1. We make sure that we have actually discovered a quad (4 connected sides).
2. We calculate the ratio of the edges (length/breadth).

By doing simple measurements on an iPad password box, we can reasonably accurately say that the ratio of length to breadth on the box should be about 13. (We can give some margin of error here, to say that the ratio should be between 8 and 14). This means that even if the picture is taken from up close, or far away, we should be able to accurately detect (and isolate) the password box. (Displayed as [2] below).



Identifying the Password Box

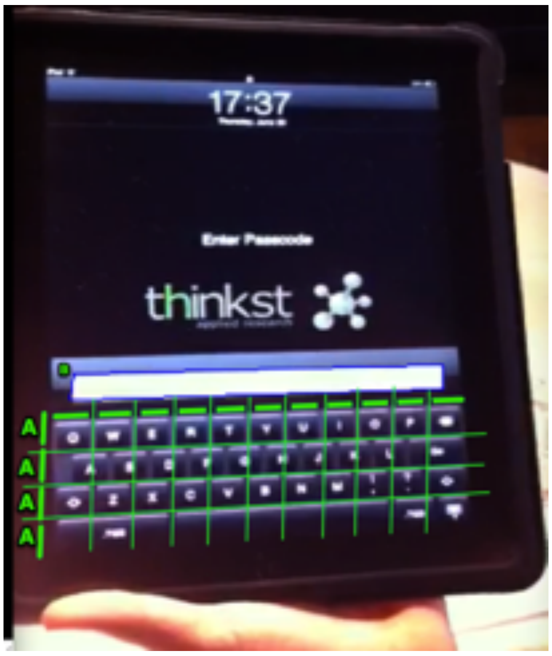
The length of the sides of the password box can now be used as a relative unit of measure. Since we have an object that can consistently orient us, and a unit of measure, we should be able to relatively easily map coordinates on the image to the keyboard.



Using the Password Box as a Reference Point.

Once we have determined that the password box is "A" high, we can measure the height of the keys, relative to "A". This means that if a key is pressed within "A" distance of of the bottom of the password box, it is in the first row. A key pressed 2*A away, must be in the second row.

By marking the size of the keys relative to "A" and relative to the left most edge of the password box (B), we can also calculate columns:



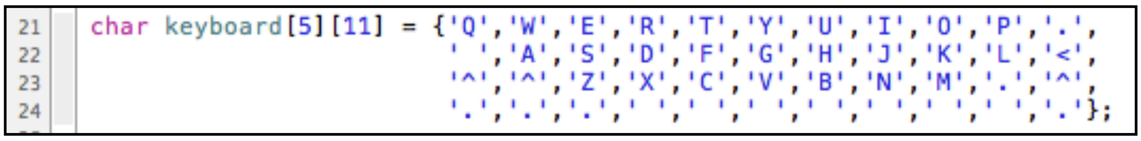
Using the Password Box as a Reference Point.

So if we have a key-press on position (X,Y), if the lowest point of the password box is (A,B) and if the height of the password box is "a", we can obtain the row, by simply saying:

row = (Y - B)/a;

(We obtain the row first, since we need to make an exception for row2 to account for the alphabet offset).

We then use the derived row and col to index our 2d keyboard array:



```
char keyboard[5][11] = {'Q','W','E','R','T','Y','U','I','O','P','.',
                        ' ','A','S','D','F','G','H','J','K','L','<',
                        '^','^','Z','X','C','V','B','N','M','.','^',
                        '.','.','.',' ',' ',' ',' ',' ',' ',' ','.'};
```

Simple keyboard 2d array

A hit on "**keyboard[3][2]**", would therefore return "**Z**". Since this relative lookup is done for every detected key-press, we effectively end up determining the position on the keyboard, relative to the password box at the point at which the key was pressed. This works to overcome the natural movement of the device.

The End Result

The end result is an app called shoulderPad, that returns the typed password, after observation (and openCV translation). A brief demonstration of shoulderPad, with full debugging, can be seen here: http://www.youtube.com/watch?v=f_iK9gYuHDg

Is the vector purely local?

Video can be captured using surveillance footage, or even long distance lenses fairly easily.

iPhone meets SLR Lenses ?

A Simple Fix

By default, the iPad makes use of a simple 4 Digit Pin. A security conscious user is able to change this 4 digit pin to a full password by adjusting the "***Simple Passcode***" setting under General preferences.

An option can be added to this setting to disable keyboard highlighting for passwords. (Even if not a default setting, it would give paranoid users the option).

Conclusion

Shoulder surfing is an old problem and most vendors have taken steps to mitigate this attack. In an attempt to provide feedback to users, current mobile devices take 2 security steps backwards leaving us less secure than we were in the past.