Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.26, NO.4, Aug. 2016 ISSN 2288-2715(Online)
http://dx.doi.org/10.13089/JKIISC.2016.26.4.857

가상 온스크린 키보드를 이용한 비밀번호 입력의 취약점 분석*

보 부 르,[1†] 김 혜 진,[1] 이 경 희,[2‡] 양 대 헌[1]
[1]인하대학교, [2]수원대학교

Analysis on Vulnerability of Password Entry Using Virtual Onscreen Keyboard*

Bobur Shakirov,[1†] Hyejin Kim,[1] KyungHee Lee,[2‡] DaeHun Nyang[1]
[1]Inha University, [2]The university of Suwon

요 약

패스워드 기반 인증 시스템은 키 로그 모니터링을 통한 정보 유출 사고에 위협받아 왔다. 최근, 이를 예방하기 위한 한 방안으로 화면 상 가상 키보드를 이용한 키 로깅 방지 방법이 널리 사용되고 있다. 그러나 이러한 가상 키보드 또한 중대한 취약점을 내포하고 있으며, 그 중 대표적인 약점은 마우스 커서의 자취 추적을 통해 쉽게 비밀번호와 같은 주요 정보가 드러날 수 있다는 점이다. 이에 본 논문에서는 가상 키보드의 취약점을 확인하고, 이를 공격할 수 있는 가상의 공격 시나리오와 패스워드를 도출하는 방법을 제시했다. 이 논문에서 제안하는 기법의 성능 입증을 위한 예시로, 한 가상 키보드에 대한 공격과 패스워드 딕셔너리를 이용한 크래킹 실험을 진행하였고, 그 결과를 분석하였다.

ABSTRACT

It is a well-known fact that password based authentication system has been threatened for crucial data leakage through monitoring key log. Recently, to prevent this type of attack using keystroke logging, virtual onscreen keyboards are widely used as one of the solutions. The virtual keyboards, however, also have some crucial vulnerabilities and the major weak point is that important information, such as password, can be exposed by tracking the trajectory of the mouse cursor. Thus, in this paper, we discuss the vulnerabilities of the onscreen keyboard, and present hypothetical attack scenario and a method to crack passwords. Finally to evaluate the performance of the proposed scheme, we demonstrate an example experiment which includes attacking and cracking by utilizing password dictionary and analyze the result.

Keywords: virtual keyboard, onscreen, password cracking

I. Introduction

Since last few decades, the phenomenon of e-commerce has been developing at high speed providing a variety of online services, such as online banking, online shopping, e-tickets, etc, to clients. However, attackers have been widely using Keyloggers to gather sensitive data and passwords of users in e-commerce. A significant number of personal computers used for online transactions also fueled

Received(05. 02. 2016), Modified(07. 12. 2016), Accepted(08. 04. 2016)

* 이 논문은 2016년도 정부(교육부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임(2014R1A1A2059852)

† 주저자, shbobur2@gmail.com
‡ 교신저자, khlee@suwon.ac.kr(Corresponding author)

the growth of Keyloggers. Keylogger is some software or hardware that monitors users' input to detect and get private crucial data such as usernames, passwords, emails, text messages or another type of data [1]. To defeat the Keyloggers and provide secure authentication, some e-banking systems are providing onscreen keyboards for users to enter sensitive data, for example, passwords. The onscreen or virtual keyboard is a software program that allow users to enter data using the mouse [2].

Nevertheless, even virtual keyboards suffer from some vulnerabilities that adversary can take advantage of:

- **Screenshot capturing.** Trojans, that are capable of capturing a screenshot at each mouse click and then send them to the adversary, can cause to being leaked of user's password during online banking account authentication.
- **Over shoulder surfing attack.** In this type of attack, an adversary steals passwords or other crucial information of the user by staying behind the user and looking over victim's shoulders while the user sits in front of the computer, therefore it requires the physical participation of attacker. This type of attack can be conducted, moreover, by using a physically hidden camera[2].
- **Videotape recording.** The only difference of this attack from the previous one is no camera is required. An attacker installs some malware to victim's computer that provides a videotape of password entering process to attacker.
-

In this paper we show another password cracking idea and design one algorithm, as an example, to crack one particular type of virtual keyboard, provided by some bank to its clients. By our idea, if an adversary can get the coordinates of mouse clicks on the virtual keyboard area, he has a great chance to generate a set of passwords, which consist of several wrong passwords with an actual password. It is possible to apply the presented idea to variety types of virtual keyboards.

Organization of paper: Section 2 briefs researchs about virtual keyboards which are resistant to some particular types of described attacks. Sections 3 consists of the detailed description of proposed idea, attacking scenario demonstration and the algorithm. Analysis and evaluation of algorithm are given in Section 4 followed by conclusion in Section 5.

II. Approaches of virtual keyboards

There are "more secure" virtual keyboards, which are actually aimed to defeat some type of attacks. In [2] the "anti-screenshot virtual keyboard" was presented. The idea is, when mouse cursor moves on some key all the keys within the row are replaced by some special character, as shown in Fig. 1. In the same way, when the user clicks the key, all keys in the virtual keyboard would be replaced with special character for a few seconds. It makes the screenshots useless in the case if attacker captures them.

Proposals of [3] and [4] are identical, Fig. 2. The main idea for both proposals is

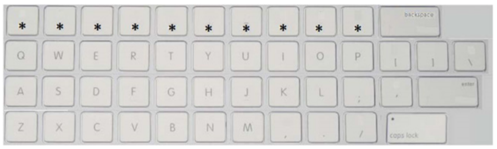


Fig. 1. Anti-screenshot virtual keyboard. Keys within the row are replaced by special character.

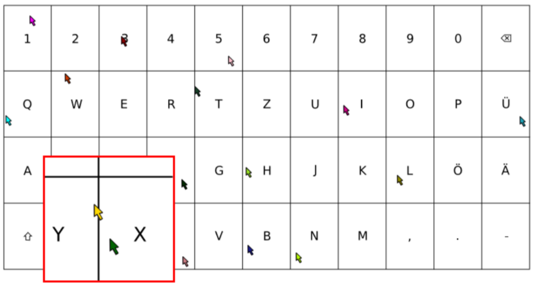


Fig. 2. Virtual keyboard with fake cursors

to use a specific number of fake cursors, besides the actual one, in order to hide input on the virtual keyboard from onlookers. Because the fake cursors move randomly and differently from the actual cursor, the user can identify it while attackers have difficulties to do so.

III. Methodology

3.1 Proposed method

The main idea of our password cracking algorithm is, having the coordinates of mouse clicks on virtual keyboard: 1. Generate candidate passwords and 2. Decrease the number of candidate passwords as many as possible.

Virtual Keyboard. To demonstrate the idea, password cracking algorithm for one specific type of virtual keyboard, which is provided by anonymous bank named W, was developed. The keyboard is presented in Fig. 3.

To switch to upper case, a user should

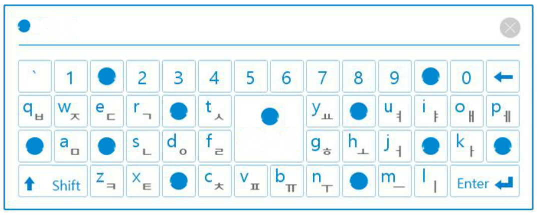


Fig. 3. Virtual onscreen keyboard of W Bank, Korea

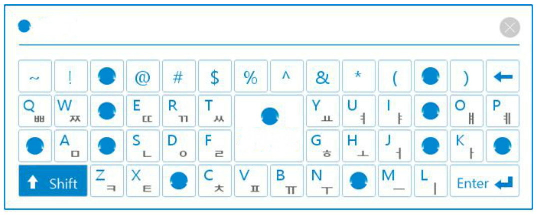


Fig. 4. Virtual onscreen keyboard with "Shift" pressed

click "Shift" key on the virtual keyboard, and, after, virtual keyboard changes in place to as Fig. 4.

Properties of the keyboard are as following:

- Each row of the keyboard has two "blank" buttons, as shown in Fig. 5;
- Each time when the user invokes the keyboard, two "blank" buttons change their positions within the row, independently from other rows;
- "blank" buttons are clickable, however, they are considered as wrong data;
- "blank" buttons are not changed from their positions after each mouse click;
- "blank" buttons move symmetrically to each other within the row;
- Functional keys – "Shift", "Enter" and "Backspace" – are not changed from their positions, therefore "blank" buttons cannot stay at their positions.

In Fig. 6, all possible cases of the first row of the virtual keyboard are presented and it clearly showes that two blank keys appear symmetrically.

Following the collecting and processing procedure is to generate candidate passwords using the coordinates. We take

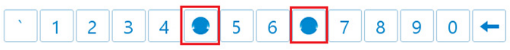


Fig. 5. Blank buttons are maintained with red rectangles.

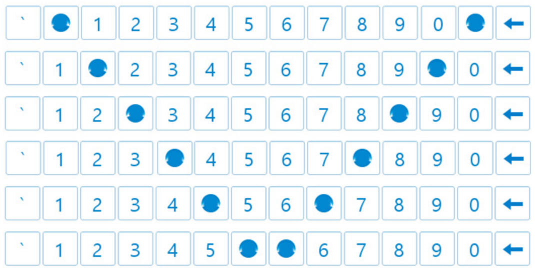


Fig. 6. All possible cases of 1st row of the virtual keyboard

advantage of two properties of the keyboard:

- "blank" buttons appear symmetrically.
- Each key position can contain at most two different actual keys or "blank" buttons, except key positions at 2, 13, 15, 20, 21, 26, 27, 31, 32, 36, 38, 42, 43, 47, which can contain only one actual key or "blank button". The blank button cannot stay at the first position.

We have the fact that, because "blank" button can stay at six different positions in a row (because of symmetric property, we will mostly talk about one-half of a row), the first and second rows of virtual keyboard have at most six different views for each one, while the third and fourth rows have by 5 for each.

For a given set of coordinates *C*, Algorithm goes through as follows:

1. Define a matrix for keyboard with 8 rows (4 lower case rows + 4 upper case rows) as a default keyboard: all blanks are at leftmost and rightmost positions, surrounding all keys;
2. Modify the matrix: change the position of "blank button" in all rows if its position exists in the *C*;
3. Go through each row of matrix and move "blank button" to each possible, not conflicting with the given coordinates, position, to find all possible cases of each row (maximum 3 cases, as described in Section 4);
4. Find all permutations of rows, making each resulting permutation as one keyboard layout—one possible keyboard case, and assign them to array *Keyboards*;
5. For each 〈keyboard case〉 in *Keyboards*: pick all coordinates in *C*, to get one candidate password, from 〈keyboard case〉.
6.

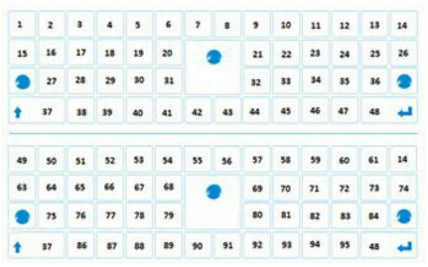


Fig. 7. Consumption

The algorithm is divided into three subproblems: password cracking—main subproblem, finding all possible candidate keyboard layouts and shifting blank buttons to appropriate positions.

3.2 Scenario design of attacking

In this section, terms "discovery" and "observation" are used in the same meaning which implies the hacking or taking the coordinates of mouse clicking on the virtual keyboard.

In the algorithm, we take one assumption: our coordinates are converted from (x,y) format to sequential format, as in Fig. 7 for convenience. All clickable buttons and buttons whose positions are changeable are numbered from 1 to 48 on lower case layout and from 49 to 95 on upper case layout. Key positions on upper case layout are numbered as "position on lower case layout + a total number of keys". Position numbers of functional keys are the same for both lower and upper registers. Note that discovered coordinates of mouse click are not sensitive to register. That is, we discover the coordinates of ″a″ and ″A″ as the same position. Algorithm differs them according to the existence of coordinates of "Shift" functional key in discovered set.

To make the suggested scenario work, the key problem we have to solve is to figure out how we can inject the code into the website. When it comes to gaining the coordinates of the cursor clicking, it is not a huge problem if you utilize tools, such as Javascript libraries. The code only contains functions that observe the cursor clicking and handle the clicking events, and is injected into the page. Eventually, the method that we are applying here is a typical Code Injection Attack—one of the most common way to hack.

In this regard, we designed a custom-made web browser extension plugin based on Chrome browser for an experiment. In this experiment, 'Content-Scripts', one of the API Chrome provides, was implemented to inject Javascript files into the web page which is specified in the configuration of the extension. When a user makes an access to W Bank log-in page, the malicious extension injects a mouse-click observing code into the page. And after the user opens and clicks on a virtual keyboard which consists of the HTML objects, 'img' and 'div', this injected code obtains and adds the coordinates of the cursor to an array in sequential order. If the user finishes typing his/her password and clicks on a login button, the injected code sends these coordinates and user's ID to attacker's server. Then, the server converts valid x-y coordinates into keyboard numbers from 1 to 48 and saves them as a file format so that the password cracking program can use it. As you can see in Fig. 8, with this strategy, the attacker can catch the key numbers in the password and its sequential order.

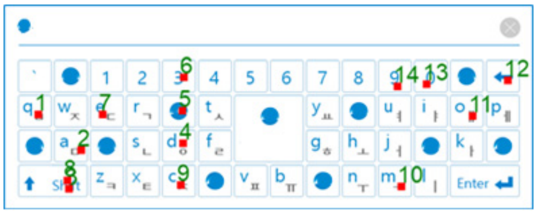


Fig. 8. Result of mouse tracking extension

To clarify the possibility in real-life circumstance, we applied this methodology to W Bank web security system. To commence the attack, there were two possible conditions that the extension can be injected into the website - with or without installing required security plugins of W Bank. First, to log in into the website on Chrome browser, installing the security plugins was avoidable on the user's choice. In this case, there was no obstacle to succeed log-in and send the coordinates to the attacker's server. On the other hand, even after installing the required security plugins and the attempt to send the code to the server was blocked by the plugins, it was possible for the attacker to bypass the sending process and still figure out the coordinates.

Although in this paper, we only operated an experiment with Chrome browser, this attack can be achieved on the other browsers' extension plugins, such as Component Object Model of Internet Explorer and Plugin of FireFox, because these browsers all equip extensions that are similar to one used in Chrome browser.

3.3 Algorithm 1: Main Part

First of all, we build candidate keyboard layouts by **Algorithm 1**, in order to obtain candidate passwords.

Main - password crack - algorithm's variables are given in Table 1.

We define matrix $Dflt\_Kbd[8][12]$ as a default keyboard. This template is built once and is not going to be changed anymore. All candidate keyboards are generated on the basis of this "default keyboard". In "default keyboard", there are 2 types of key positions: "empty" and "non-empty". The key position and its symmetric one in the row are "non-empty" if its position exists as the coordinate in the set C. For instance, if $k \in C$ is clicked by the user, then it claims that the key in the position k is not a blank button. Hence, we mark the position k as non-empty, which corresponds to "the position is not for blank button".

Initially, all keys are the type of empty. "for" loop in the lines 3-10 scans all key positions to define their type. In line 12, **Algorithm 2** is used to find out all candidate keyboard layouts and the result is assigned to array $Keyboards$.

The cycle in the lines 13 to 27 "types" all candidate passwords and includes all into the set P. Going through all given coordinates, in lines 16-25, all pressed key characters are picked from "$Keyboards$" and appended to new_pwd. "if" condition in lines 17-23 checks each coordinate whether it is a "functional" key.

Table 1. Variables of **Algorithm 1.**

Variable	Description
C	set of coordinates of mouse click
$Dflt\_Kbd$	default keyboard: Template for all candidate keyboards. Values are either *empty* or *non_empty*.
$Keyboards$	set of candidate keyboard layouts
$shift$	used to keep track of the "Shift" functional key
empty	type of a key.
non_empty	type of a key.

Algorithm 1. Password crack algorithm.

INPUT: set of coordinates C
OUTPUT: set of candidate passwords P

```
1. Initialize $empty$ = 0, $non\_empty$ = 1, $blank$ = "-";
2. Initialize $Dflt\_Kbd[8][12]$ to $empty$, $Keyboards[\ ]$, $P=\{\ \}$;
3. for $row=1$ to 8 do
4.   for $key=1$ to $Dflt\_Kbd[row].length/2$ do
5.     if $Dflt\_Kbd[row][key].coord$ exists in $C$ or $Dflt\_Kbd[row][key].symmetric.coord$ exits in $C$ then
6.       $Dflt\_Kbd[row][key]=non\_empty$;
7.       $Dflt\_Kbd[row][key].symmetric=non\_empty$;
8.     end if;
9.   end for;
10. end for;
11.
12. $Keyboards$ = Find out possible candidate keyboard layouts ($Dflt\_Kbd$);
13. for $i=1$ to $Keyboards.length$ do
14.   $shift=0$;   // default - Shift is not pressed
15.   $new\_pwd$="";
16.   for $coord=1$ to $C.length$ do
17.     if $C[coord]==14$ then //Backspace
18.       delete($new\_pwd.last$);
```

```
19.     else if $C[coord]==37$ then //Shift
20.       if $shift==0$ then $shift=48$
21.       else $shift=0$;
22.     else if $C[coord]==48$ then break
              all cycles; //Enter
23.     else
    $new\_pwd=new\_pwd+Keyboards[i][coord+shift]$
    ;
24.     end if;
25.   end for;
26.   $P\cup new\_pwd$;
27. end for;
28. return $P$;
```

3.4 Algorithm 2: Generating all possible candidate keyboard layouts

In second step, **Algorithm 2** is used to generate all candidates for keyboard. Its input is main — template keyboard. Its variables are listed in Table 2.

Initially, blank buttons are on the leftmost and rightmost positions of the rows. In the lines 4-14, all blank buttons in Kbd_Case are shifted to empty positions if their positions are non-empty. In the lines 6 to 11 all keys are shifted to one position and in lines 12 and 13 blank buttons are placed at appropriate positions. Lines 16 to 20 go through each row of Kbd_Case. "loop" in 17th and 19th lines are repeated until it collects all candidates of the given row into $PossibleRowCases$.

Lines 21 to 32 find all permutations of rows of the keyboard, where each element of permutation is one line of the keyboard, hence, each of the permutations represents one candidate keyboard. Algorithm gathers all permutation into array $Keyboards$. In line 22 all candidates of the first row are assigned to $Keyboards$, so at the moment, the first lines of candidate keyboards are found. "for loop" in Line 23, constructs permutations from 2nd to 8th elements. Process is as following: "for loop" in line 25 repeats as many times as the length of $Keyboards$ to the moment. "for loop" in line 27, increments the "length" of permutation by concatenating each candidate of a row with existed permutation, moreover, including new permutation as new element into $Keyboards$. In line 30, first element—preexisted permutation is deleted from $Keyboards$, after it was used to make new permutation.

In the lines 34 to 43, the keys, with unchangeable positions and not considered at the permutation, inserted into their fixed positions.

Table 2. Variables of **Algorithm 2**.

Variable	Description
$PossibleRowCases$	Collects all possible permutations of each row separately.
Kbd_Case	used to find a new candidate for a row. In any moment each of its rows contain only one candidate.
$Number\_Of\_Keys$	used to switch the register.
$blank$	used as blank button
$Keyboards$	collection of all possible candidate keyboards.

Algorithm 2. Find out possible (candidate) keyboard layouts

INPUT: $Dflt\_Kbd$

OUTPUT: $Keyboards$ - set of all possible (candidate) keyboard layouts

```
1. Initialize $Keyboards[]$,
   $PossibleRowCases[8][]$ to "",
   $Number\_Of\_Keys=48$;
```

```
2. Initialize $empty = 0$, $non\_empty = 1$,
   $blank =$"-", $len = Dft\_Kbd[i].length$;
3. Initialize $Kbd\_Case[8][] =$
   [["-1234567890-"], ["-qwertyuiop-"],
   [-asdfghjk-"], ["-zxcvbnml-"],
   ["-!@#$%^&*()-"], ["-QWERTYUIOP-"],
   ["-ASDFGHJK-"], ["-ZXCVBNML-"]];
4. for $i = 1$ to 8
5.   $j = 2$;
6.   while $Dflt\_Kbd[i][j-1] == non\_empty$
7.        and $j < len/2$ do
8.     $Kbd\_Case[i][j-1] = Kbd\_Case[i][j]$;
9.
       $Kbd\_Case[i][j-1].symmetric =$
                 $Kbd\_Case[i][j].symmetric$;
10.    $j = j+1$;
11.  end while;
12.  $Kbd\_Case[i][j-1] = blank$;
13.  $Kbd\_Case[i][j-1].symmetric = blank$;
14. end for;
15. // Step iv
16. for $row = 1$ to 8 do
17.   do loop
18.
    $PossibleRowCases[row].append(Kbd\_Case[row])$
    ;
      // append - adds new element to
    the end of array
19.   while doNextSwap
      $(Kbd\_Case[row], Dflt\_Kbd[row])$;
20. end for;
21. // Step v
22. $Keyboards = PossibleRowCases[1]$;
23. for $row = 2$ to 8 do
24.   $k = Keyboars.length$;
25.   for $i = 1$ to $k$ do
26.     len = $PossibleRowCases[row].length$
27.     for $case = 1$ to len do
28.       $Keyboards.append(Keyboards[1] +$
                 $PossibleRowCases[row][case])$
29.     end for
30.     delete$(Keyboards[1])$
31.   end for
32. end for
33. // Insert keys to each keyboard layout
34. for $i = 1$ to $Keyboards.length$ do
35.   $Keyboards[i]$.insert(1, "`")
36.   $Keyboards[i]$.insert(14, "|")// Backspace
37.   $Keyboards[i]$.insert(37, "|")     // Shift
38.   $Keyboards[i]$.insert(48, "|")     // Enter
39.   $Keyboards[i]$. insert ( 1 +
    $Number\_Of\_Keys$, "~")
    // first position in Upper case
    keyboard
40.   $Keyboards[i]$. insert ( 14 +
    $Number\_Of\_Keys$, "|") // Backspace
41.   $Keyboards[i]$. insert ( 37 +
    $Number\_Of\_Keys$, "|") // Shift
42. end for
43. return $Keyboards$
```

3.5 Algorithm 3: Shifting blank buttons to next possible position

The algorithm below is used to shift blank buttons efficiently. Its inputs are a row of keyboard — *case_row* and the corresponding row of default keyboard — *default_row*. If the shift occurs, then the algorithm returns true and false otherwise. Variables of **Algorithm 3** are listed in Table 3.

Loop in the lines 3 to 5 finds the

Table 3. Variables of **Algorithm 3**.

Variable	Description
default_row	any row of template keyboard
case_row	row to be changed
change	returned as a result
middle	position of the key in the middle of the given row
prev_pos	previous position of blank button in *case_row*
next _pos	new position of blank button in *case_row*

position of the blank in the $case\_row$ and assigns it to $prev\_pos$. Next position is looked for in $default\_row$. Next efficient position of the blank button is the empty position after one non-empty if it exists. When several empty positions stay next to each other in one row, if we put blank to each one of them and take all as a different cases of the row, consequently, several keyboards differentiated only with that row give us, unfortunately, repeated candidate passwords. Therefore, we need only one candidate from that row's cases.

Algorithm 3. doNextSwap

INPUT: one row of $Dflt\_Kbd$ — $default\_row$, and one respective row of Kbd_Case — $case\_row$

OUTPUT: true — if blank in $case\_row$ shifted to next appropriate position, false —if shift is not possible more

```
1. Initialize $empty = 0$, $non\_empty = 1$, $blank =$“-“;
2. Initialize $change = false$, $middle = default\_row.length/2$,
   $prev\_pos = 0$, $next\_pos = 0$;
3. while not $case\_row[prev\_pos] == blank$ do
4.     $prev\_pos = prev\_pos + 1$;
5. end while;
6. $next\_pos = prev\_pos + 1$;
7. while $default\_row[next\_pos] \equiv empty$ and $next\_pos < middle$ do
         // find next non-empty position in default keyboard row
8.     $next\_pos = next\_pos + 1$;
9. end while;
   // find next empty position in default keyboard row
10. while $default\_row[next\_pos] == non\_empty$ and $next\_pos < middle$ do
11.     $next\_pos = next\_pos + 1$;
12. end while;
13. if $next\_pos > middle$ then
14.     $change$ =false;
15. else
16.     shift blank to position $next\_pos$;
17.     shift second blank to position $next\_pos.symmetric$;
18.     $change$ =true;
19. end if;
20. return $change$;
```

3.6 How to reduce the number of candidates?

The first step of the idea has been described. Next step is to reduce the number of candidate passwords to the minimum, to 1. Even though goal is to reduce the number of candidates to 1, due to the fact that most systems give 3 attempts to try entering password and some give 5 attempts, we can make notions: if the number of candidates is 4 or 5, it is a “good” result; if it is 2 or 3 then it is a “better” and if that number is 1 then it is the “best” result. Therefore, in spite of that we aim to get the best result, better or good results are both acceptable.

The solution for the 2nd step would be observing authentication process more than once. Naturally, coordinates of the same password on different layouts give several set of password candidates: the content of the sets are different, but they should consist of at least one or a few common candidate passwords. The intersection of these sets results in a set of fewer candidate passwords. There is one case where the intersection is not useful and it is described in “Evaluation” section.

For instance, adversary observed 2 times the same user. The first set of coordinates is $\{26\ 27\ 28\ 28\ 16\ 25\ 19\ 30\ 37\ 2\}$. The second is $\{25\ 28\ 29\ 29\ 17\ 24\ 19\ 30\ 37\ 2\}$ and algorithm gave following sets of candidate password:

$$S_1 = \{password!, passworf!, passwotd!, passwotf!\}$$

$$S_2 = \begin{Bmatrix} password!, psddworf!, oasseird!, osddeirf!, \\ oasseitd!, osddeitf! \end{Bmatrix}$$

Intersection is:

$$S = S_1 \cap S_2 = \{password!\}$$

The resulting set consists of one candidate which is true password.

So the second step is observing several times.

IV. Evaluation

4.1 Result

To estimate our algorithm, we conducted one experiment. At the beginning of this section, the experiment is introduced, at the end, worst case analysis of algorithm is given.

In order to evaluate efficiency of the algorithm, we attempted to find out how many candidate passwords could be given by first time observation and by second, third, fourth and so far. To do so, we obtained leaked passwords list from RockYou [5], chose several different virtual keyboard layouts (one layout for one observation) and wrote one script to convert given text password to set of coordinates. The list of passwords is filtered to exclude the passwords with the content of non-existing character in our virtual keyboard. A total number of passwords in the list is 1,024,560. Into one text file, namely keyboards_map, we wrote coordinates of each key for chosen keyboards (e.g. A:position number, B:position number, C:position number, etc.). The script takes one password from the list and one set from keyboards_map and returns the set of coordinates of the corresponding password. Then, each set is given to algorithm. The number of candidates in the first observation (from one keyboard layout), after the intersection of the first and second observation sets, intersection of the 1st, 2nd and 3rd, finally, intersection of the 1st, 2nd, 3rd and 4th observation sets are recorded. Results are below.

To describe the graphs, we use notions defined in section 3.6.: good, better and best result. Four lines with different colors are observations. The horizontal line is the number of candidate passwords in the set and vertical line corresponds to the number of actual passwords used in the experiment. If the point is marked at the coordinates 4 (horizontal) and 600,000 (vertical) then it means that for 600,000 passwords, there are ≤4 candidate passwords for each password. Indeed, using the notions, we can say: 600,000 passwords with good result includes also passwords with better and best results. When we say about the number of a better result, actually, this number includes passwords with the best result too.

Fig. 9, demonstrates one experiment. After first observation, we got best results for only 58,195 passwords (~5.6%) and

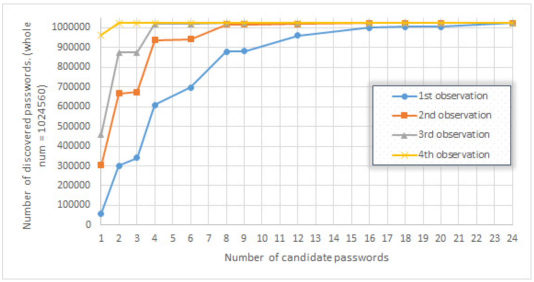


Fig. 9. Result from the first experiment

better results for 340,225 passwords (~33.2%). However, good results were nearly 60% - 608,276 passwords. Second observation, as expected, improved all best, better and good results to 30%, 65%, and 91%, respectively. In the third observation, good results were 99.5% (1,019,860 passwords), better results were 85% (875,040) and best results - 45% (459,818). After fourth observation for 1,023,749 (99.9%) passwords result was better and best for 963,501 passwords (94%).

In another experiment, a different set of randomly chosen virtual keyboards are used. Results are as following: 1st observation, approximately 40% (405,184 passwords) - good, 186,542 passwords - better and 22,129 passwords - best results. The second observation was enough to get 99% good results (for 1,018,911 passwords) and 86% - for 883,930 passwords best result.

4.2 Some mathematical analysis

For mathematical analysis, we do not estimate speed or storage usage of the algorithm, but we want to find out a possible number of candidates for a password. As we found, the number of candidates depends solely on the positions of password characters on the row of the virtual keyboard. Each used row may cause to multiplying the number of candidates from 1 to 3 times. We define 3 types of rows: 1-case row, 2-case and 3-case rows. The rows type of 1-case may give only one candidate, 2-case rows may multiply 2 times the number of candidates and 3-case rows may multiply 3 times. Graphical views of all types are given in Fig. 13.

We define k_i as the number

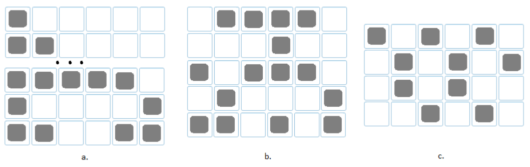


Fig. 10. (a) some possible cases of 1-case rows. (b) some possible cases of 2-case rows. (c) all possible cases 3-case rows. Gray rectangles are used keys in a row. Only half of row is shown.

corresponding to the type of row i. The value of k_i ranges in $[1,2,3]$. If row_i belong to 1-case rows then $k_i=1$, if row_i is 2-case row then $k_i=2$ and $k_i=3$ if row_i is 3-case row. The number of candidates is:

$$N_{candidates}=\prod_{i=1}^{8}k_i$$

In the best case, one observation can give only one candidate, which is an actual password. Theoretically, in the worst case candidates maybe up to $3^8=6561$. However, in our experiments, this number ranged between $[1,256]$. And we found that in all cases it is easy to exclude some wrong candidates of the password. If in the password, some meaningful words are used, it is easy to guess which candidate is more close to being true password.

V. Conclusion

In this article, we have shown how we exploited the vulnerability of virtual keyboards. One particular type of virtual keyboard was analyzed and described from a security perspective, as an example. The introduced idea can be applied to another type of virtual keyboards. It does not require any hardware or physical participation of adversary.

To sum up, these types of onscreen keyboards are vulnerable to our attack and those systems who are using virtual keyboards should rethink their security approaches.

References

〔1〕 Mehdi Dadkhah and Mohammad Davarpanah Jazi, "A novel approach to deal with keyloggers," Oriental Journal of Computer Science & Technology, Vol. 7, no. 1, pp. 25-28, Apr. 2014

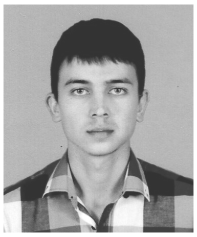
〔2〕 Ankit Parekh, Ajinkya Pawar, Pratik Munot and Piyush Mantri, "Secure authentication using anti-screenshot virtual keyboard," International Journal of Computer Science Issues, Vol. 8, Issue 5, pp. 534-537, Sep. 2011

〔3〕 Alexander De Luca, Emanuelvon Zezschwitz, Laurent Pichler and Heinrich Hussmann, "Using fake cursors to secure on-screen password entry," Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 2399-2402, Apr. 2013

〔4〕 Keita Watanabe, Fumito Higuchi, Masahiko Inami, and Takeo Igarashi, "CursorCamouflage: multiple dummy cursors as a defense against shoulder surfi ng," SIGGRAPH Asia 2012 Emerging Technologies, no. 6, pp. 15-16, Nov. 2012

〔5〕 https://wiki.skullsecurity.org/Passwords

〈저자소개〉



보 부 르 (Bobur Shakirov) 학생회원
2015년 6월: Tashkent University of Information Technologies 컴퓨터공학과 학사
2015년 9월~현재: 인하대학교 컴퓨터정보공학과 석사과정
〈관심분야〉 암호이론, 네트워크 보안, 보안 프로그래밍, 인증 프로토콜

김 혜 진 (Hyejin Kim) 학생회원
2016년 2월: 인하대학교 컴퓨터정보공학과 학사
2016년 3월~현재: 인하대학교 컴퓨터정보공학과 석사과정
〈관심분야〉 암호이론, 생체인증, 네트워크 보안

양 대 헌 (DaeHun Nyang) 종신회원
1994년 2월: 한국과학기술원 과학기술대학 전기 및 전자공학과 학사
1996년 2월: 연세대학교 컴퓨터과학과 석사
2000년 8월: 연세대학교 컴퓨터과학과 박사
2000년 9월~2003년 2월: 한국전자통신연구원 정보보호연구본부 선임연구원
2003년 2월~현재: 인하대학교 컴퓨터정보공학과 교수
〈관심분야〉 암호이론, 암호 프로토콜, 인증 프로토콜, 무선 인터넷 보안

이 경 희 (KyungHee Lee) 정회원
1993년 2월: 연세대학교 컴퓨터과학과 학사
1998년 8월: 연세대학교 컴퓨터과학과 석사
2004년 2월: 연세대학교 컴퓨터과학과 박사
1993년 1월~1996년 5월: LG소프트(주) 연구원
2000년 12월~2005년 2월: 한국전자통신연구원 선임연구원
2005년 3월~현재: 수원대학교 전기공학과 부교수
〈관심분야〉 바이오인식, 전보보호, 컴퓨터비전, 인공지능, 패턴인식