CS645 Spring 2012 \ Project 3

Ronny Abraham, Ariel Stolerman

1)

This works to the advantage of the good guys.

Let's say computers have become one order of magnitude faster - they are now twice as fast (in binary).

The good guys only need to add one more bit to their key, and since their operations are linear in the length of the key, they only see a linear increase in processing time (which is partially compensated by the fact the computers are twice as fast).

The bad guys, on the other hand, have their operations exponential in the length of the key; by adding one more bit this doubles the range of the key. So for example, in brute forcing, the computer may be now twice as fast, but the number of possible keys is also double, which means the 'bad guy' operations are doubled in computing time.

2)

There are a few ways one could protect against key loggers. Generally speaking, there are two types of key loggers: software-based and hardware-based.

Software-based key loggers run at a low level such as under the OS, in the kernel, or as malware in the API. A software logger can be more sophisticated than just recording keystrokes, it also knows about the context (which part of an application the user is typing into).

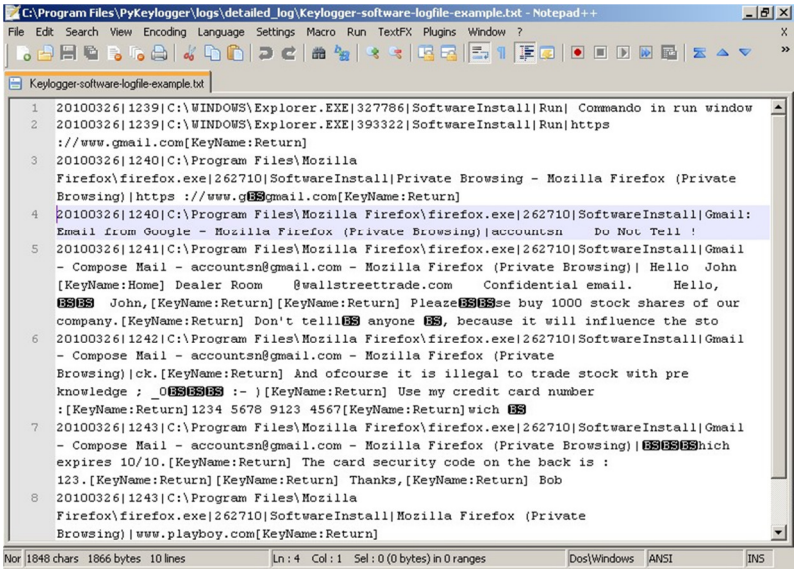


Image 1 – example of software-based key logger

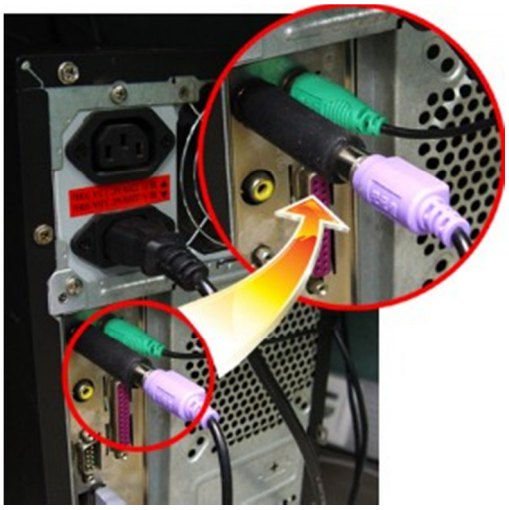


Image 2 – A hardware-based key logger

Hardware-based key loggers are a piece of hardware that intercepts the actual keystroke signals going between the keyboard and a computer. There is also another form of hardware key loggers – these are firmware-based in the form of malware in the BIOS. In either case, only physical keystrokes are recorded in the logger.

Ways to mitigate a key logger attack:

1. The question states that an adversary can only log keystrokes, which implies there is no knowledge about the session. If this is true, a user could add diffusion and confusion just by typing a character, then switching to another application (let's say a text editor) type some characters, then switch back, type some more, and continue this way. An attacker only sees a string and has to figure out which characters belong to the target application and which ones are bogus. If we look at an n-length string consisting of real and bogus characters, there are 2^n sub groups which only one of them contains the correct passwords. This makes a brute-force attack exponential in the length of the string. While this method works, it seems a little unfriendly for a user (Note: in order for it to work, switching to another application should be done with the mouse to obfuscate the switch, and not keyboard shortcuts like Alt+TAB in windows).
2. Now let's assume the attacker does know about context. In this case switching applications and adding more characters will not work, since our adversary knows which characters belong to the target application and which are just noise. To overcome this, we can have keystroke interference embedded into our target application. In Keystroke interference, the software injects programmatic keystrokes in between the key strokes. Similar to the previous method, this makes a brute-force attack exponential in the number of the string seen by the key logger. This method also protects against knowledge of context, and is user friendly: the users just type their passwords as they normally do. Note that while this works for software-based key logger, it does not protect against hardware-based key loggers. Hardware-based key logger only records real keystrokes - programmatic interference key strokes are not logged.
3. Another way to deal with key loggers is by having some kind of form-manager or password-completion feature in software, in which the user just clicks in the password field, and the application automatically fills in the password. While this may work both for software or hardware key loggers, this has significant shortcomings. First, we need to deal with the first time a password is stored, or each time a password needs to be changed. We need to develop a way to store\change a password without using a keyboard. This is also very unsecure since anyone could just login the system by clicking the password filed (which defeats the idea of a password).
4. Another way is to use a form of input that is not a keyboard. In class we talked about using dongles as passwords, or other biometric readings (i.e. a finger scan, or eye scan). We can add to that more forms of input such as speech recognition, or hand gesture reading.
5. A user could also use a virtual keyboard (supplied by the target application or the OS) and then click with the mouse on the characters of the password. While this may work for hardware-based key loggers, software-based key loggers can still read the key-events sent to the OS (since they usually run at a low level). Mapping the keys to some other values not always works since the key logger can capture images of the mouse and crack the cipher. But then again, the question is stats the logger can only read key strokes so any of these suggested methods will work.
6. Another facilitation of the mouse is opening some random web page or document with many characters, and tediously copy each desired character and pasting it to the password field (can even be not in order). The idea is to take advantage of the key-logger blind spot – the mouse – to hide what characters are copied and where they are pasted.

3)

1. $\boldsymbol{h(x) = f(x) \circ g(x)}$ **is collision resistant**:

Proof:

Assume by contradiction $h(x)$ is not collision resistant, then it is easy to find x_1, x_2 such that:

$$x_1 \neq x_2 \wedge h(x_1) = h(x_2)$$

If we substitute we get:

$$f(x_1) \circ g(x_1) = f(x_2) \circ g(x_2)$$

By isolating $f(x_1), g(x_1), f(x_2), g(x_2)$ we have found x_1, x_2 such that:

$$f(x_1) = f(x_2), g(x_1) = g(x_2)$$

Which means we have just described an algorithm to find a collision for f and g, but this contradicts the fact that one of f or g is collision resistant (we assumed the output of f and g has a known number of bits, because a hash function will take as input a string of any length, and return a fixed-length hash value). □

2. $\boldsymbol{h(x) = f\big(g(x)\big)}$ **is NOT collision resistant**:

Proof:

Suppose f is collision resistant but g is not (it is a valid assumption because we do not know which of f or g is collision resistant and which is not). This means we can easily find x_1, x_2 such that:

$$x_1 \neq x_2 \wedge g(x_1) = g(x_2)$$

Therefore:

$$h(x_1) = f\big(g(x_1)\big) = f\big(g(x_2)\big) = h(x_2)$$

Hence it is equivalently easy to find a collision for h, thus h is not collision resistant.

□

3. $\boldsymbol{h(x) = f\big(g(x)\big) \circ g\big(f(x)\big)}$ **is NOT collision resistant**:

Suppose f is collision resistant and g is the constant function $g \equiv C$ (where C is some constant). Let x_1, x_2 be two different input strings, then:

$$h(x_1) = f\big(g(x_1)\big) \circ g\big(f(x_1)\big) = f(C) \circ C$$

And:

$$h(x_2) = f\big(g(x_2)\big) \circ g\big(f(x_2)\big) = f(C) \circ C$$

Therefore:

$$h(x_1) = h(x_2)$$

For any x_1, x_2, thus h is not collision resistant (it is the constant function $h \equiv f(C) \circ C$).

□

4)

To estimate the probability of collision, we rely on the assumption that the given hashing algorithms (MD5 and SHA-1) are pseudorandom (thus correlating outputs to inputs is infeasible). Under this assumption, for any given file, any of the possible hashes are equally likely. This also means we assume that the hashing events are independent of each other.

To estimate the probability of collision, we first need to get estimations of the total number of files in the world. First, the total number of computers in the world is approximated to be 1.1 billion [1]. It was hard to find an estimation of the number of files on an average computer, but according to [2], an average home computer should contain somewhere between 100,000 and 200,000 files. The number of files on a server however should probably be much higher, so we can generously estimate the number of files per computer to be 1 million (very generously…). Therefore the total number of files we assume exists in the world is $10^6 \cdot 10^9 = 10^{15}$.

Next we examine the problem like the birthday paradox. This is simply the balls-and-bins problem: given N balls and M bins, what is the probability when randomly assigning balls to bins that some bin has at least two balls. In this problem the balls are the set of all files in the world and the bins are the set of possible hash values (which depends on the algorithm). If we map the problem to the birthday problem itself, the files are the set of people, the possible hash values are the set of possible birthdays, and a collision is two people having the same birthday.

Looking at the birthday problem [3], we can generalize the probability that there is no collision, and take the complement. Let F denote the number of files and H the number of hash values:

$$\Pr[collision] = 1 - \frac{F!\binom{H}{F}}{H^F} = 1 - \frac{F!\dfrac{H!}{F!\,(H-F)!}}{H^F} = 1 - \frac{H!}{(H-F)!\,H^F}$$

We can use an upper bound (after all we want to know the highest probability of a collision, as it is the worst case scenario) as shown in [4] to ease the computation:

$$\Pr[collision] \leq \frac{F(F-1)}{2H}$$

For MD5, $H = 2^{128} \cong 3.4 \cdot 10^{38}$ therefore:

$$\Pr[MD5\ collision] \cong \frac{10^{15}(10^{15}-1)}{2 \cdot 3.4 \cdot 10^{38}} \cong \frac{10^{30}}{7 \cdot 10^{38}} = \frac{1}{7} \cdot 10^{-8} \quad (\cong 10^{-9})$$

For SHA-1, $H = 2^{160} \cong 1.4 \cdot 10^{48}$ therefore:

$$\Pr[SHA-1\ collision] \cong \frac{10^{15}(10^{15}-1)}{2 \cdot 1.4 \cdot 10^{48}} \cong \frac{10^{30}}{3 \cdot 10^{48}} = \frac{1}{3} \cdot 10^{-18} \quad (\cong 10^{-19})$$

Therefore even under the generous assumptions on the number of files, looks like for both MD5 and SHA-1 it is very unlikely to find a collision under the assumptions.

[1] http://howmanyarethere.net/how-many-computers-in-the-world/

[2] http://www.answerbag.com/q_view/1640550

[3] http://en.wikipedia.org/wiki/Birthday_problem

[4] http://www.cs.brown.edu/courses/csci1510/reference/goldwasser_bellare_notes.pdf

5)

1.

Approach for solving the problem:

The ASCII encoding system:

The extended ASCII encoding system provides 8-bit (or 7-bit in the non-extended) encoding of characters, which means each byte represents a character. For this assignment, where most characters probably represent English letters (and perhaps punctuation and digits, especially for the code ciphers) – and all definitely represent only readable characters, it means most bytes should be encoded in the first half of the ASCII range (i.e. 0x0 0 0x7F).

This can be even further narrowed to \t, \n, \r and characters with code 0x20 – 0x7E (" " through "~"). This can be utilized later when trying to figure out for $c_i \oplus c_j$, where $c_i \in P_i, c_j \in P_j$ are plaintext characters, the probable range of c_i, c_j. This is further explained under "predictability-of-languages" below.

Another convenient property of ASCII code is that corresponding uppercase and lowercase English letters differ by exactly 0x20, meaning that for finding matches using uppercase or lowercase won't matter for any English letter strings.

Properties of XOR (Exclusive-or):

The relevant properties of XOR are:

Let A, B, C be some n-bit strings and $0^{(n)}$ n-bit a string of zeros:

- $A \oplus 0^{(n)} = A$
- $A \oplus A = 0^{(n)}$
- $\oplus$ is invariant to permutations (i.e. $A \oplus B \oplus C = C \oplus A \oplus B = \cdots$ all other permutations)

From these properties we can immediately see that for two ciphers C_1, C_2 generated from two plaintexts P_1, P_2 with the same key K:

$$C_1 \oplus C_2 = (P_1 \oplus K) \oplus (P_2 \oplus K) = P_1 \oplus P_2 \oplus K \oplus K = P_1 \oplus P_2 \oplus 0^{(n)} = P_1 \oplus P_2$$

Therefore if we retrieve the correct plaintext-pairs, we get some mixture of them that can be the basis of extracting the plaintexts themselves.

In addition, we can also utilize the fact that for $K_1 \neq K_2$: $C_1 \oplus C_2 = P_1 \oplus P_2 \oplus (K_1 \oplus K_2)$, where, in case K_1 and K_2 are random keys, $K_1 \oplus K_2$ is like a random key itself. This can be utilized for identifying which plaintext pairs were encrypted with the same key, as the plaintext "mix" (i.e. $P_1 \oplus P_2$), given the rather narrowed range its characters can be in, will probably look different than when you throw in a key (i.e. $K_1 \oplus K_2$). This is demonstrated in part 2.

Predictability of languages:

Human and computer languages have properties than can utilize the result of $P_1 \oplus P_2$ to extract the actual plaintexts. For instance word frequencies in the English language – like articles ("the", "an"), or preserved words in computer language – like "if", "while", "int" (depending on the language). These words are good candidates to start looking for (relying on word-frequency statistics we find online). How is it done?

Say we have the word "while" in one of P_1 or P_2. If we XOR a sequence of "while"s with $P_1 \oplus P_2$, the corresponding position in the result should be a meaningful string. For instance, assume the word "`while`" appears in P_1, and in the same position in P_2 appears the word "`that `", then we will discover it by XORing:

$P_1: x_1x_2x_3 \dots while \dots x_n$

$P_2: y_1y_2y_3 \dots that \dots y_n$

$P_1 \oplus P_2: z_1z_2z_3 \dots z_i \dots z_{i+4} \dots z_n$

$\Rightarrow P_1 \oplus P_2 \oplus whilewhilewhile \dots while = u_1u_2u_3 \dots that \dots u_n$

Of course since we don't know the position of "while" in P_1 we will have to try all possible positions:

```
whilewhilewhile…while
0whilewhilewhil…ewhil
00whilewhilewhi…lewhi
…
```

Where 0 pads the beginning (or we can pad in a cyclic manner with "", "e", "le" etc.).

Eventually we might get enough hints that, combined with the properties we known on the plaintexts (quote of Shakespeare etc.) may help reveal P_1 and P_2 (we won't know originally who's who, but it doesn't matter, as long as we obtain both plaintexts).

To make things a bit easier, we can disregard unreadable characters (as mentioned in "The ASCII encoding system" above), to narrow down candidates (practically: map all non-readable results to some constant character in order to "clean" the XOR results, and make it easier to find meaningful strings).

<u>Other factors that can help solve the problem</u>:

The hints, of course – once enough words are discovered, Googling them can help.

In addition, for specific hints:

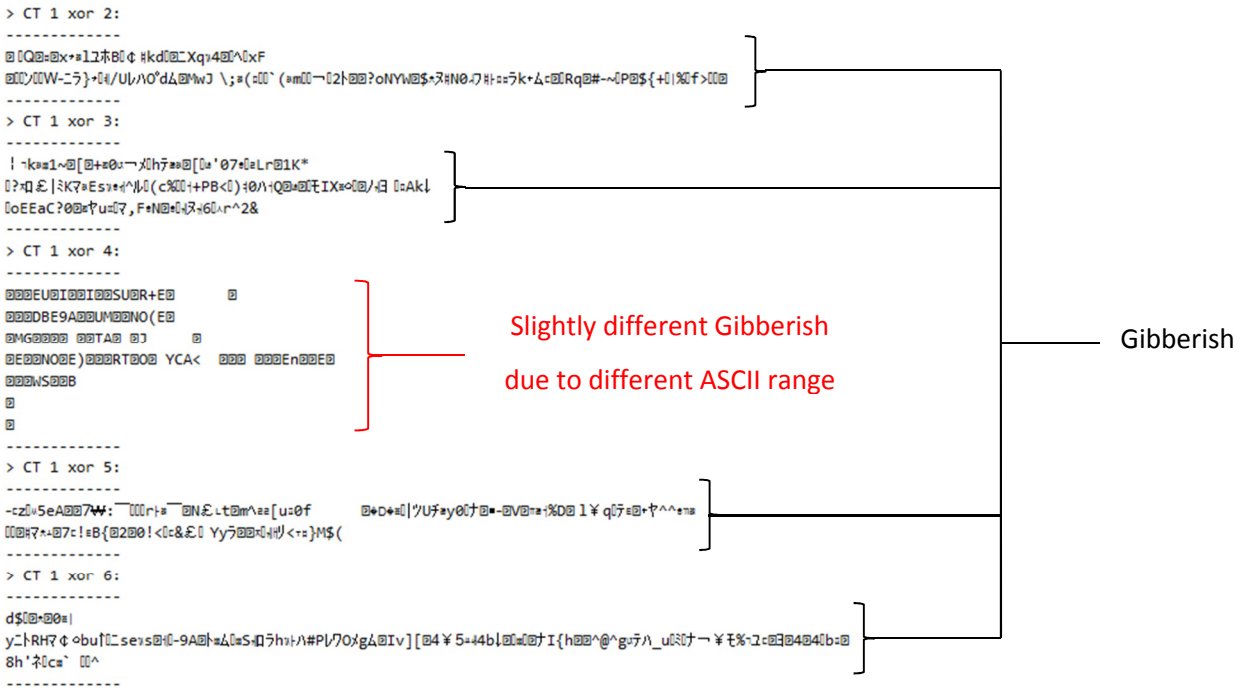
- Shakespeare quote – start with "to be or not to be" (probably the most famous quote) – in retrospective, this is too easy, but taking advantage of human predictability (i.e. not making this assignment TOO hard but not too easy as well) – should have looked for substrings of the rest of this famous quote (which turned out to be the one).
- Code – in retrospective it would have been smart to try out strings of the codebase in project 1 + 2 (thus taking advantage of human predictability again – not stretching too far to get a code sample for this assignment ☺).
- Using the keyword "lyrics" when Googling helped a bit.

2.

The ciphertext pairs are:

- C_1 with C_4
- C_2 with C_6
- C_3 with C_5

Demonstrating the difference between the XORs:



3. The plaintexts are:

P_1 and P_4	From Rick Astley's *Never Gonna Give You Up* song lyrics (with a missing "e" at the end): Never give you up Never gonna let you down Never gonna run around and desert you Never gonna make you cry Never gonna say goodby From The Beatles' *Yellow Submarine* song lyrics: And our friends are all aboard. Many more of them live next door. And the band begins to play. We all live in a yellow submarine
P_2 and P_6	Douglas R. Hofstadter (starts with a space): an eerie type of chaos can lurk just behind a facade of order and yet, deep inside the chaos lurks an even eerier type of order From *The Hunger Games*: Welcome! Welcome! Happy Hunger Games and may the odds be ever in your favor! Now, before we begin, we have a very special film...
P_3 and P_5	From Shakespeare's *To be or not to be* (there's an extra space at the end): Whether tis nobler in the mind to suffer the slings and arrows of outrageous fortune, or to take arms against a sea of troubles Code taken from project 1 + 2 sploit's stubs with slight modifications: int main(void){ char *args[3]; char *env[1]; args[0] = TARGET; args[1] = "hi there"; args[2] = NULL; env[0] = NULL; if 0>execve.