Das Taggen von Texten

Inhalt

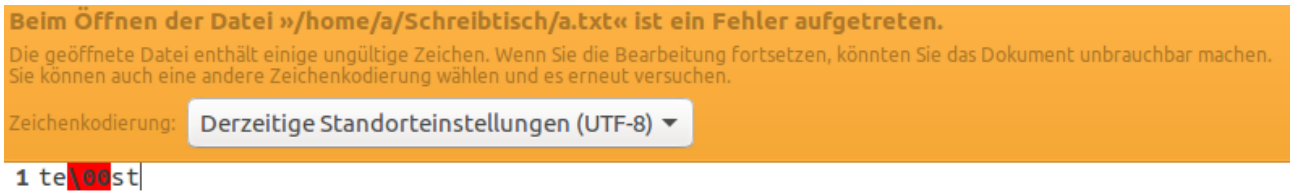
September 2017

Vorwort

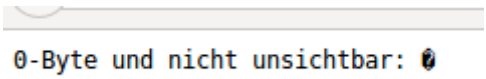
Dieses Dokument soll beschreiben, wie Texte jeglicher Art mit einer eindeutigen Kennung versehen werden können. Egal ob 100 Zeichen oder eine Million Zeichen. Texte wie beispielsweise geheime- oder lizenzierte Dokumente geraten immer mal wieder an die Öffentlichkeit. Einmal veröffentlicht ein abtrünniger Staatsangestellter geheime Dokumente und ein anderes Mal kopiert jemand vertrauliche Texte aus einem Chat, Account-Daten aus einer Notiz oder gar ein ganzes Buch-Manusskript. Wenn Texte einmal im Internet stehen lässt sich selten ermitteln welche der Personen die Zugriff auf den entsprechenden Text hatten ihn nun auch veröffentlicht hat. Bei digitalen Bildern kann man kleine Pixel innerhalb des Bildes verändern. Bei PDF-Dateien kann man Wasserzeichen Verstecken oder auch sichtbar einsetzen. Bei geliehenen/gestreamten Videos lässt sich ein Millisekunden kurzes, nicht wahrnehmbares Bild einsetzen das Beispielsweise den Namen des Käufers oder dessen Kundennummer enthält, usw.. Doch was macht man mit reinem Text, Texten die immer wieder kopiert werden, z.B. auf Twitter veröffentlicht, per Mail kopiert, weiter gesendet und dann wiederum bei Facebook geteilt? Wenn man auch solche Texte mit Kodierungen/Wasserzeichen versehen könnte um dessen ursprünglichen Besitzer auch nach vielem Kopieren, Zerschneiden und Abändern des Textes per Messenger, Schreibprogramm oder Social-Media noch ausfindig machen könnte dann hätte man ein Interessantes Werkzeug um Verräter, Raubkopierer und ungebetene Plaudertaschen zu identifizieren. Verräter und ungebetene Plaudertaschen nahm ich hierbei als Anreiz nach einer Lösung für dieses Problem zu suchen. Zumindest nach einer die dem Wasserzeichen in PDF-Dateien und Identitäts-Snapshots in Videos gleich kommen könnte. In diesem Dokument beschriebe ich meine Lösung für dieses Problem, wie man Texte mit eindeutigen Kennungen versehen kann ohne dass diese beim Kopieren, Beschneiden, Bearbeiten oder Versenden, Teilen usw. diese Kennung verlieren.

Methodik

Möchte man Texte jeglicher Art, egal ob Word-Dokument, Reiner Text oder beispielsweise Nachrichten auf einer Social-Media-Plattformen wie Facebook dauerhaft mit Kennungen versehen, so muss man mit dem Text selbst arbeiten. Informationen wie Kennungen die nicht im Text selbst enthalten sind gehen beim bloßen Kopieren u. Bearbeiten des selbigen verloren. Schließlich kann man einfach den Text selbst kopieren und weitergeben/veröffentlichen anstatt dessen umliegende Informationen wie Beispielsweise das Impressum einer Website von der man den Text hat. Ebenso helfen hierbei auch keine Urheberrechtsangaben im Dokument selbst, denn diese muss man nicht mit-kopieren. Man kann alle Stellen des Textes in denen etwas auf den Urheber oder Besitzer bzw. ursprünglichen Empfänger hindeutet einfach weglassen/entfernen. Dementsprechend sollte eine Kennung innerhalb des Textes zum Ersten nicht frei sichtbar sein und zum Zweiten mehrmals enthalten sein damit bei Änderungen des Textes nicht jeglicher Eintrag zur Kennung zerstört werden oder verloren gehen kann. Hierfür benötigt man ein Zeichen, welches beim Lesen/Ansehen des Textes vom Betrachter nicht gesehen werden kann und dennoch von Schreibprogrammen sowie Texteditoren, dem Browser usw. nicht bemängelt wird. Dafür kommen augenscheinlich einige Zeichen in Betracht. Beispielsweise das 0-Byte-Zeichen. Schreibt man dieses Zeichen in einen Text, so sollte an der stelle wo das 0-Byte-Zeichen steht nichts sein. Soweit klappt das auch ganz gut, doch öffnet man einen solchen Text beispielsweise in einem Schreibprogramm, dann meldet dieses einen Fehler, weil das 0-Byte-Zeichen nicht zum Standard-Zeichensatz gehört mit dem das Programm seine Texte und Dokumente lesen möchte.

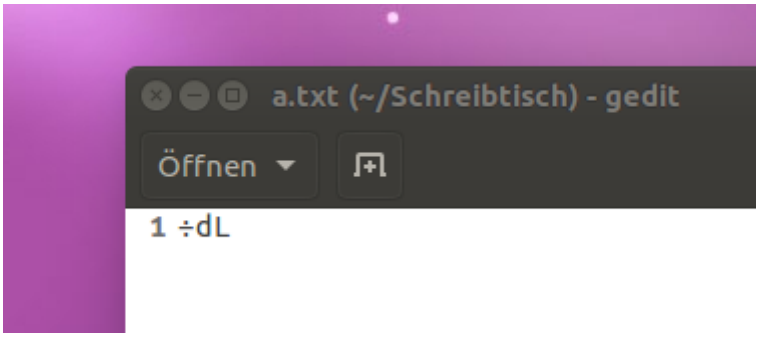


UTF-8 ebenso wie UTF-16, welches UTF-8 enthält sind die am weitesten verbreitetsten Zeichensätze der Welt. Mit UTF-8 lassen sich Zeichen verschiedenster Art darstellen. Man kann deutsche Texte mit Umlauten ebenso wie kyrillische, fernöstliche oder afrikanische darstellen. Dementsprechend ist UTF-8 der Text-Standard im IT-Bereich! Zeichen in UTF-8, welche man eigentlich nicht sehen können sollte werden für gewöhnlich entweder als mysteriöses Viereck oder Fragezeichen dargestellt.



Sollte keines von beidem zutreffen, dann nimmt das Zeichen für gewöhnlich derart auf die darauf folgenden Zeichen Einfluss, dass die Existenz des unsichtbaren Zeichens offensichtlich wird. Dementsprechend gibt es eigentlich kein Zeichen, welches sich ernsthaft in Texten verstecken lässt. Außer einem, dem UTF-8 Zeichen selbst! Dieses Zeichen nutzt man als BOM (Byte Order Mark). Bom-Angaben werden eistens am Anfang eines Textes eingefügt um diesem eine bestimmte Zeichenkodierung auf zu erlegen. Beispielsweise kann am Anfang einer Text-Datei eine BOM-Angabe gemacht werden, welche das Zeichenset für den gesamten Text festlegt. Diese BOM-Angabe wird dann nicht als Zeichen dargeboten sondern lediglich als Angabe für die zu nutzende Zeichenkodierung genutzt. Der Leser sieht diese BOM-Angabe also nur in so fern, dass die Text-Ausgabe nun in einer bestimmter Zeichenkodierung gemacht wird. Allerdings kann man nicht einfach eine beliebige BOM-Angabe setzen und dann darauf vertrauen, dass diese ewig bestehen bleibt. Denn kopiert ein Leser nur einen Teil des Textes, so kann das BOM am Anfang der Text-Datei beim vielen versenden und Teilen schnell verloren gehen!

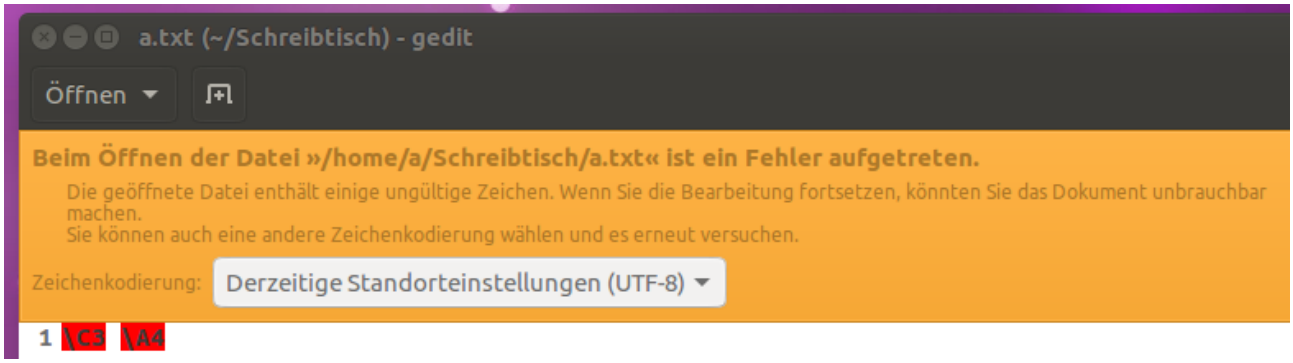
Hinzukommend akzeptieren viele Text-Editoren/Schreibprogramme keine eigenen BOM-Angaben sondern erzwingen ihre eigene Standard-Angabe die für gewöhnlich UTF-8 ist. Hier ein Beispiel:



Beim Versuch das UTF-1-Zeichen (Hex: F7644C) am Anfang der Text-Datei zu setzen gibt der gängige Linux-Texteditor „gedit“ dieses Zeichen aus, anstatt den Text in UTF-1-Kodierung aus zu geben. Das selbe geschieht übrigens nicht nur in diesem Text-Editor sondern auch in vielen anderen gängigen Schreibprogrammen, Browsern, usw. auf verschiedenen Betriebssystemen.

UTF-8 kann zwar so ziemlich jedes gängige Zeichen darstellen, verwendet dafür allerdings immer den selben Zeichenstamm. Anstatt diesen Zeichenstamm stetig um die entsprechenden benötigten Zeichen zu erweitern interpretiert UTF-8 Zeichen, welche nicht zu seiner Zeichenpalette gehören, indem es mehrere Zeichen seiner normalen Zeichenpalette zusammensetzt. Der Buchstabe „a“ gehört beispielsweise zur normalen Zeichenauswahl im UTF-8 Zeichensatz, der Umlaut „ä“ hingegen nicht. Dieser wird dann mit 2 Zeichen (Hex: C3A4) dargestellt welche direkt hintereinander geschrieben werden. Ein Beispiel in PHP:
`file_put_contents('a.txt', "\xC3\xA4")`

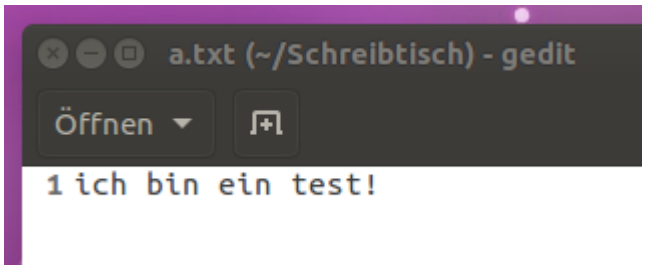
In diesem Beispiel wird beim öffnen der Text-Datei „a.txt“ ein ä ausgegeben. Schreibt man die zwei Zeichen jedoch nicht zusammen sondern setzt beispielsweise ein Leerzeichen zwischen diese, so kann UTF-8 diese zwei Zeichen nicht mehr als Eines verstehen:
`file_put_contents('a.txt', "\xC3 \xA4")`



Das selbe geschieht auch beim BOM. Auch die BOM-Zeichen setzen sich aus mehreren einzelnen Zeichen zusammen und werden dann als eines Interpretiert.

Die Hexadezimalzusammensetzung für UTF-8 lautet EF BB BF. Setzt man diese 3 Zeichen hintereinander dann erhält man das UTF-8 Zeichen:
file_put_contents('a.txt', "\xEF\xBB\xBFich bin\xEF\xBB\xBF ein test!")

Dieses UTF-8-Zeichen wiederum bleibt, egal an welcher Stelle unsichtbar im Text enthalten, da das UTF-8-Zeichen innerhalb der UTF-8 Kodierung nicht unbekannt ist:



Nun haben wir ein Zeichen, welches gut und gerne in den gängigen Texteditoren sowie Messengern und Browsern innerhalb von Texten eingefügt werden kann ohne dass dieses angezeigt wird. Bleibt nun nur noch die Frage wie man mit einem einzigen Zeichen eine Vielzahl verschiedener und jeweils einzigartiger Kennungen erstellt. Schließlich benötigt man wenigstens 2 Zeichen um mehr als nur eine Hand voll einzigartiger Kennungen zu erstellen. Hätte man beispielsweise nur eine 0, so könnte man verschiedene Kennungen nur erstellen indem man immer mehr Nullen an einander reiht: Kennung 1: 0, Kennung 2: 00, Kennung 3: 0000, Kennung 4: 0000, usw.. Möchte man aber mehr Kennungen zur Verfügung haben so benötigt man mindestens 2 Zeichen: Kennung 1: 0101, Kennung 2: 0001, Kennung 3: 0011, usw.. Um dieses Problem zu lösen können wir den entsprechenden Text selbst nutzen. Das UTF-8 Zeichen ist dabei das zu versteckende Zeichen (nennen wir es „x“), alle anderen Zeichen dienen als Platzfüller, so wie im Beispiel der Kennungen bei dem z.B. die 1 für das UTF-8-Zeichen stehen könnte und die Nullen für alle anderen Zeichen. Die Position des UTF-8 Zeichens innerhalb des Textes kann dementsprechend genutzt werden um massenhaft eindeutige Kennungen zu erstellen. Ein Beispiel: Am Anfang einer Kennung steht das UTF-8-Zeichen und die Positionen weiterer UTF-8-Zeichen innerhalb nachfolgender 6 Zeichen ergeben eine eindeutig Kennung:
ohne Kennung: abcdef
Mit Kennung 1: \xEF\xBB\xBFab\xEF\xBB\xBFcd\xEF\xBB\xBFef
Mit Kennung 2: \xEF\xBB\xBFabc\xEF\xBB\xBFd\xEF\xBB\xBFef
usw...

Oder mit „X“ als UTF-8-Zeichen ausgedrückt:
Kennung 1: XabXcdXef
Kennung 1: XabcXdXef

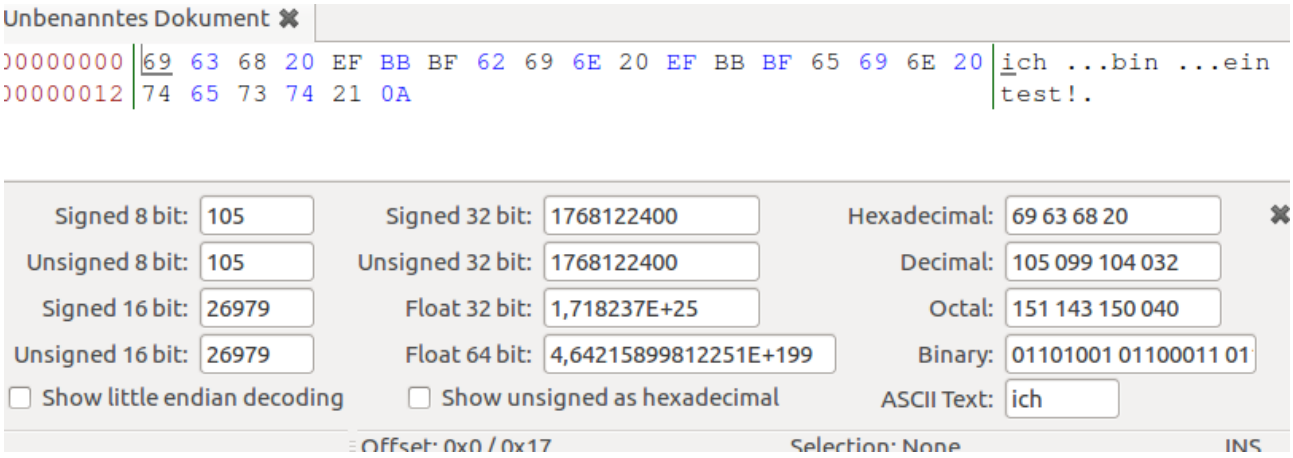
Auf diese Weise lässt sich mit Hilfe der Position des jeweiligen UTF-8-Zeichens innerhalb eines festgelegten Radius eine Eindeutige Kennung einsetzen. Je nach Größe des Radius, welcher für die Kennung vorgesehen ist können nun unzählige Kennung verwendet werden! Nutzt man Beispielsweise 6 UTF-8-Zeichen innerhalb von 20 Zeichen des Textes (Wobei jede Kennung mit dem UTF-8-Zeichen beginnt damit der Anfang einer Kennung auch erkannt werden kann.), so hat man 11628 verschiedene Möglichkeiten/Kennungen zur Verfügung! Nun muss man sich nur noch eine dieser Kennungen aussuchen und wiederholend in den entsprechende Text einbauen. Sollte ein Teil des Textes editiert werden oder verloren gehen dann bleiben immer noch genügend andere Eintragungen in anderen Teilen des Textes erhalten in denen man die eindeutige Kennung ausfindig machen kann.

Anwendung

Um nun diese Methode ernsthaft ein zu setzen ist es wichtig zu ermitteln welche Programme den Text ohne das UTF-8-Zeichen anzeigen und welche es mitnehmen. Wie bereits erwähnt wird das UTF-8-Zeichen nur in auffallend wenigen Programmen angezeigt und kann dementsprechend effektiv im Text versteckt werden. Test-Text:
ich \xEF\xBB\xBFbin \xEF\xBB\xBFein test!

Beispielsweise in diesen Programmen/Web-Seiten bleibt das Zeichen verborgen:
- gedit
- Libre Office
- Windows 10 Notepad
- Notepad++
- Chromium/Google Chrome
- Mozilla Firefox (Bei beliebten Web-Seiten wie z.B. Facebook)
- Opera Browser
- Safari (Bei beliebten Web-Seiten wie z.B. Facebook)
- Microsoft Edge (Bei beliebten Web-Seiten wie z.B. Facebook)
- Mac OS X TextEdit
- Thunderbird
- iOS Mail
- Windows 10 Mail
- Microsoft Outlook
- OpenOffice (Bei standard Datei-Formaten wie z.B. .odt, .doc, .rtf, usw..)
- Microsoft Office/Microsoft Word (Bei standard Datei-Formaten wie z.B. .odt, .doc, .rtf, usw..)
- WordPad (Bei standard Datei-Formaten wie z.B. .odt, .doc, .rtf, usw..)

Allerdings gibt es auch Editoren/Programme mit denen man das Zeichen sehen kann. Allerdings sind diese oft sehr speziell und eben nur für den professionellen Gebrauch im IT-Bereich gedacht. Bless ist ein solches Programm, er ist ein Hex-Editor mit dem man bequem (ohne eigene Scripte zu schreiben) jeglichen Inhalt einer Datei (Egal welchen!) einsehen kann:



Nun wissen wir, dass das UTF-8-Zeichen so ziemlich bei allen vorhaben verborgen bleiben kann. Jetzt braucht es nur noch eine große Liste mit fertigen Kennungen und ein Script, welches diese Kennungen in Texten versteckt und wiederfinden kann. Für ersteres erstellt man einfach eine Passwortliste mit dem Zeichensatz 0, 1 oder z.B. „x“ sowie Leerzeichen. Ich selbst habe „x“, Leerzeichen genutzt mit Kombinationen betreffend einer Länge von 20 Zeichen. Anschließend muss man nichts weiter machen als diese Liste zu Filtern. Dieses Script erstellt eine entsprechende

Liste und filtert nach allen Einträgen welche mit einem „x“ beginnen und genau 6 mal „x“ enthalten (C++):

```
#include <algorithm>
#include <iostream>
#include <fstream>
#include <math.h>

using namespace std;

int main() { string s="x              xxxxx"; string sold=s; long i=0; long snn=pow(2, s.length());
string* arrayxlist=new string[snn]; arrayxlist[0]=s; long iii=1;
while (i<snn) { bool val=next_permutation(s.begin(), s.end()); if (val!=false) { std::string sx=s; int
sxl=sx.length(); if (sx.substr(0, 1)=="x") { string sxa=""; int ii=0; while (ii<sxl) { if (sx.substr(ii,
1)=="x") { sxa+="x"; } ii++; } if (s==sold) { i=snn; } else { arrayxlist[iii]=s; } iii++; } i++; } }
string newlistoutx=""; i=0; while (arrayxlist[i].length()>0) { newlistoutx+=arrayxlist[i]+"\n"; i++; }
ofstream myfile; myfile.open("xnew-dic.txt"); myfile<<newlistoutx; myfile.close(); return 0; }
```

Nach der Nutzung des Scripts sollten noch 11628 Kombinationen übrig bleiben. Das dürfte für den Anfang genügen! Selbstverständlich lässt sich die Anzahl der verfügbaren Kombinationen mithilfe von mehr „x“ oder einer größeren Gesamtlänge bis ins unendliche steigern.

Nun zu jenem Script, welches eine Kombination in einen Text einfügen und wieder herausfiltern soll (Ruby-Script):

```
#---conf---
fwname='text.txt'
frname='out.txt'
xcode="xxxx  x  x          "
xjumpmax=500 #plus 2x length of xcode +2 +count of x in xcode
#----------

#---write---
xread=0; xwrite=0
if ARGV.length!=1
print "usw w=write or r=read\nusage: *.rb w\n"; exit
else
if ARGV[0].clone.to_s().downcase=='r'
xread=1
end
if ARGV[0].clone.to_s().downcase=='w'
xwrite=1
end
end

xtext=File.read(fwname); xtextl=xtext.length; xcodel=xcode.length; xcodell=xcode.gsub(' ',
'').length; xjumpmin=xcodel+2+xcodell; xjumpmint=xjumpmin*2;
if xwrite==1
i=xjumpmin+Random.new(Random.new_seed).rand(xjumpmax);
xnewtext=""; xtexti=0; xtextlx=xjumpmin+xjumpmax
while i<xtextl-xjumpmint-3

while xtexti<i
xnewtext+=xtext[xtexti].to_s; xtexti+=1
```

```
end

if i<xtextl
ii=0
while ii<xcodel
if xcode[ii]=='x'
xnewtext+="\xEF\xBB\xBF"
end
xnewtext+=xtext[i].to_s
i+=1; ii+=1
end
end

xtexti=i; i+=xjumpmint+Random.new(Random.new_seed).rand(xjumpmax)
end

while xtexti<xtextl
xnewtext+=xtext[xtexti].to_s; xtexti+=1
end

File.write(frname, xnewtext); print 'insert "'+xcode+'"\"\nfile size before:
"+File.stat(fwname).size.to_s+" byte\nfile size now: "+File.stat(frname).size.to_s+" byte\n";
end
#-----------

#---read----
if xread==1
print "search for \""+xcode+"\"\nblock length: "+xcodel.to_s+"\n\n"; bi=1; xtext=File.read(frname);
xtext=xtext; xcodel+=xcodel+xcodell; i=xjumpmin; xtextl=xtext.length

while i+5<xtextl
xbr=0
while i+5<xtextl
if xtext[i].to_s=="\xEF\xBB\xBF"
xbr=1; break
end
i+=1
end

if xbr==1
xcoder='x'; i+=1; ii=0; while ii<xcodel
if xtext[i].to_s=="\xEF\xBB\xBF"
xcoder+='x'
else
xcoder+=' '
end
i+=1; ii+=1
end
end
xcodergx=xcoder.gsub('x ', 'x')
if xcodergx.length>xcode.length
xcodergx=xcodergx[0...xcode.length]
```

```
end
if xcoder.to_s=='0'
xcoder='???'
end

if xcodergx==xcode
print bi.to_s+": found!\n"
else
print bi.to_s+': "'+xcodergx.to_s+'\" unknown!\n"; i+=xjumpmin-5
end
bi+=1
end
end
#-----------
```

Zu beachten ist hierbei der Anfang des Scripts, wo sich die Konfiguration befindet. Dort trägt man bei der Variable „fwname“ jene Datei ein, welche den reinen Text enthält, welcher mit den versteckten Kennungen ausgestattet werden soll. „frname“ wiederum ist der Dateiname in den der bearbeitete Text ausgegeben wird. Beim Aufspüren einer Kennung nutzt das Script ebenfalls den Text aus der Datei von „frname“. „xcode“ enthält jene Kennung die das Script in den jeweiligen Text einfügen soll. Zu guter letzt „xjumpmax“, diese Variable gibt an wie groß die Abstände zwischen den einzelnen Kennungen maximal seien dürfen. Wie weit die Kennungen im Text von einander entfernt sind wird vom Script per Zufall bei jedem ausführen neu berechnet (Damit nicht jeder bearbeitete Text/Datei haargenau die selbe Datei/Text ist.). Die maximale Länge dient nur dazu diesen Bereich des Zufalls nach belieben ein zu grenzen. Um das Script zu starten nutzt du dann den Parameter „w“ zum ändern eines Textes und „r“ zum auslesen/suchen einer Kennung. Hier ein Beispiel mit den selben Einstellungen wie im Beispiel-Script und einem 1676 Byte großen Text: