#### Pattern Developer’s Guide

Pafwert 1.5

Author: Mark Burnett (mb@xato.net)

February, 2007

**Copyright & Legal Disclaimer**

Copyright ©2003-2007, Mark Burnett, All Rights Reserved. No part of this Work, which includes this document and any related material, may be reproduced or distributed in any form or by any means without prior written permission of Mark Burnett, with the exception of code listings which may be reproduced for individual use.

Mark Burnett, the author(s), and any individual or organization involved in the production of this Work do not guarantee or warrant the results obtained from this Work.

This Work is provided AS IS and is without warranty of any kind, expressed implied, regarding this work. Always use appropriate precautions, make backups, and test thoroughly in a non-production environment before following any instructions or running any applications or scripts included with this Work.

All trademarks are either trademarks or registered trademarks of their respective companies.

# Table of Contents

Table of Contents ii

1. Introduction 1

Predictable Patterns 1

Using Pafwert Effectively 2

2. Pattern Development 4

Pattern Elements 4

Working with Password Templates 6

Creating Strong Passwords 7

3. Placeholder Functions 9

Asc 9

Chr 9

Consonant 9

Entropy1, Entropy2, Entropy3 9

EndPunctuation 10

Keyboard 10

LeftHand 10

Letter 10

LongDay 10

LongMonth 11

Now 11

Number 11

NumberCode 11

NumberPattern 12

NumRow, NumRowFull 12

Ordinal 12

Phonetic 12

Pronounceable 13

RightHand 13

Row1, Row1Full, Row2, Row2Full, Row3, Row3Full 14

Sequence 14

ShortDay 14

ShortMonth 14

Smiley 14

Space 14

Symbol 15

Vowel 15

Word(text file) 15

4. Word Modifiers 16

A 16

Bracket 16

Format 16

Hide 16

LCase 17

Left 17

Mid 17

Num2Word 17

Obscure 17

PigLatin 19

ProperCase 19

Quote 19

Random 19

RandomCase 20

Repeat 20

Replace 20

Right 20

RomanNumeral 20

Scramble 21

SentenceCase 21

Stutter 21

Swap 21

Trim 21

UCase 21

5. Pattern Examples 23

# Introduction

The Weakness of Passwords

Despite technology advances in many areas of security, passwords still very much play a central role in protecting our information systems. Yet they are often the weakest link in any security system. The problem with passwords is that users generally do not select strong passwords yet random generators do not create passwords that are easy for users to remember.

One problem with user-selected passwords is that even if they are strong, they tend to follow the same patterns after a while. Despite the creative capacity of humans, we are way too predictable. If I asked you to list the last 20 passwords you have selected, inevitably some sort of pattern will emerge.

The reason we fall into this trap is because we are told to select strong passwords yet we are told not to write them down. Therefore we use words that are meaningful to us and employ patterns so that we do not forget the password. For example, if my last password was Carrot28, it would be very easy to make my next password something like Carrot29 or 28Carrot. By following patterns, we reduce the chances of forgetting our passwords.

And that is why passwords are cracked so easily. Patterns are easily spotted and easily mimicked.

## Predictable Patterns

To make conditions worse, humans are horribly predictable. We do the same things day after day, month after month, year after year. And humans as a body are predictable as well. Many people are going to follow the same patterns in selecting their passwords. Over the years, I have collected password lists from a variety of sources. I have seen distinct patterns emerge in these lists. For example:

More people are likely to replace the letter o with the number 0 than any other letter/number replacement. When adding numbers to a password, users are more likely to add one or two numbers at the end of the password than add numbers to the beginning or in the middle. For example, a user is more likely to use the password pickle99 than 99pickle. Users are more likely to use a common word found in the dictionary than a nonsense or totally random word. Users are more likely to use a noun than any other part of speech. The number 1 appears in twice as many passwords than most other numbers. The number 2 comes in second place with 1.5 times as many occurrences. The number 8 is least likely to be used in a password. If a user has ever used a word in a password, chances are that word will be used again in some other password.

Clearly, by studying patterns in passwords, one could easily develop a very effective password cracking system based on common rules.

Why Pafwert?

We have all likely seen totally random passwords like SbnA4e#Vcm spit out by password generators. Sure, those are strong passwords, but most users must write them down to remind themselves of the password. But whenever it comes time to create a password off the top of my head, I struggle to come up with something unique that is also strong. So that is why I created Pafwert.

Pafwert generates random passwords. Pafwert passwords are usually quite strong and would be very difficult to guess. But Pafwert passwords are also easier to remember than passwords such as SbnA4e#Vcm.

Why are they easier to remember? Because Pafwert uses many of the same techniques that we employ to better remember things. It uses techniques such as patterning, repetition, rhymes, homonyms, antonyms, humor, meter, clichés, familiar words, and offensive words to create phrases that are easier to store away in our heads without writing them down.

For example, consider these five passwords created with Pafwert:

Interrogate a housefly 3 times?

Volume 8: Mitigate

c:\inetpub\wwwroot\smelly.htm

Bravo-Echo-Echo-Papa

zeus@suez.com

The first password is a totally random sentence built from various parts of speech. It takes a random verb, a random noun, a random number, and some other random elements. The result is a long password that isn't too difficult to remember.

The next password, Volume 8: Mitigate employs rhyming and meter. If you say the password aloud, you will see what I mean. The two portions each have 3 syllables and each end with the same "ate" sound. The password is 18 characters long and uses upper and lower-case letters, a number, and a symbol. Ok, maybe its not as easy to remember as Brutus4@penis.gov, but it is nonetheless a great password.

Another one of my favorite algorithms is the file path. The password c:\inetpub\wwwroot\smelly.htm was generated from this algorithm and because I work so much with Windows, it’s a structure that is easy for me to remember. The first portion, c:\inetpub\wwwroot\, is a path that many admins have already had to memorize, and the addition of smelly.htm is humorous enough to stick in our minds. Using 29 characters and a variety of character sets, it’s a password that is not likely to be cracked in any reasonable amount of time.

Next, while not one of my favorite algorithms, the password Bravo-Echo-Echo-Papa is still a pretty good password. Some may find this type of password easier to remember than others (it spells out the word BEEP).

Finally, zeus@suez.com is another variant of the e-mail address algorithm. This time the word zeus is an anagram for the word seuz. An anagram is one word that contains all the same letters as another word, but arranged differently.

Pafwert allows you to build an unlimited number of patterns using many wordlists and modifiers to come up with trillions and trillions of password combinations.

## Using Pafwert Effectively

As you generate passwords enough Pafwert passwords, you will soon recognize the limitations of a computer. The one thing I can't code is the ability for Pafwert to recognize a password that is easy for any particular person to remember. I can employ techniques for increasing the likelihood of this happening, but Pafwert may still spit out random garbage like Lo:||rte1[FHFH] and ... <!--567--> 1.11.9(rdrd. To Pafwert, those seem like very easy-to-remember patterns, but its not likely a human will think so. You may have to generate several passwords find a password that really stands out.

Remember that users should use Pafwert passwords to help a user’s own creativity. Normally you should not assign user passwords directly from Pafwert, but offer several passwords and let the user select one. A user may like most of a generated password, but decide to leave off some extra characters at the end that made it too confusing. Or, a user may take elements from several different passwords to create something that is more meaningful to them. You should encourage this practice. Keep in mind that this is software code and someone may at some time find a flaw in one of the algorithms. While I have taken many precautions to prevent that, by adding your own patterns and encouraging users to use their own creativity, you can add another level of uniqueness to your passwords.

# Pattern Development

## Pattern Elements

Pafwert passwords are based on pattern templates located in the patterns.cfg file in Pafwert’s wordlist directory. Each pattern is a simple case-insensitive text string.

A pattern consists of one or more of the following elements:

* Wordlists
* Placeholder functions
* Modifiers
* Parameters
* Qualifiers
* Selection groups
* Back references
* Static text

### Wordlists

Wordlists are the basic building blocks for passwords. Pafwert builds passwords by combining wordlist words with other elements. Relevant and clean wordlists will always produce the best passwords.

Pafwert includes a number of pre-built wordlists but you can also include any of your own lists. A wordlist file is a plain text file with a single word per line.

### Placeholder Functions

Placeholder functions are the core element of Pafwert passwords. Generally they are wordlist lookups, but may also be one of several built-in lists such as vowels. Placeholder functions are enclosed in braces and will be replaced with a random word when the pattern is processed. For example the pattern {word(noun)} will be replaced with a random noun from the nouns.txt file in the wordlist directory. See Section . for a complete reference.

### Modifiers

Modifiers are used within placeholders to enhance their capabilities. A modifier follows a placeholder function but is enclosed within the braces of the function to be modified. The following are examples of modifiers:

{Word(noun)+UpperCase}

{vowel+repeat(2)}

{Word(4-letter)+bracket}

Note that all modifiers can be also used as placeholder functions, as explained later in this section. See Section . for a complete reference.

### Parameters

Some functions and modifiers have optional or required parameters. Parameters immediately follow the function or modifier and are enclosed parenthesis; multiple parameters are separated by commas. The Word placeholder shown in the examples above takes a parameter indicating which wordlist to use to locate a random word.

Parameters do not need to be enclosed in quotes unless they contain a leading or trailing space or if you wish to pass an empty string. Below are example parameters:

{word(noun)+scramble(2)}

{word(phrases)+replace(“ “,””)}

### Qualifiers

Qualifiers allow you to increase the randomness of a pattern by weighting the use of functions or modifiers to a certain percentage point. Qualifiers immediately follow any parameters and are enclosed in brackets. Qualifiers are a number between 1 and 99 indicating the percent chance that the placeholder function or modifier will be applied. For example, the pattern {vowel[50]} indicates a fifty percent chance that a vowel will be chosen. If not chosen, the placeholder will be removed from the pattern. Below are examples of modifiers:

{word(adjective)[50]} {word(noun)}

{vowel}{consonant[20]}{vowel}

{Word(Noun)+UpperCase[50]}

### Selection Groups

Selection groups are a special placeholder function that randomly selects from a pipe-separated (“|”) list of values. The selection group will be replaced with one of the listed values. Note that items in a selection group can contain placeholder functions, including more selection groups. Below are examples of selection groups:

{a|b|c|d}

{We|He|She|They}

{{Word(noun)}|{Word(verb)}}

{www|{ftp|dns|mail}}

Note that embedding recursive selection groups allows you to make certain selection items appear at a greater frequency than others. In the last example above, there will be a fifty percent chances of selecting www and a fifty percent chance of selecting one of the other three, resulting in www appearing more than any others.

### Back References

Back references allow you to refer to the result of a previous placeholder function. Back references begin with the prefix $W followed by the placeholder to be processed. For example, the pattern {letter}{$W1} will be result in the same letter appearing twice.

Placeholder functions are numbered from right to left, starting with the deepest level placeholder. For example, the pattern {Word({verb}noun})}{EndPunctuation}will return either verb or noun as $W1, the result of the Word function as $W2, and the result of the EndPunctuation function as $W3. Note that a back reference cannot refer to a placeholder that occurs earlier in the pattern. For example, this pattern would not be valid: {$W2} {Word(noun)}.

Note that you cannot use modifiers directly on back references, but you can by enclosing the entire word with another set of braces. For example, the pattern {$W1+scramble} is incorrect. The correct pattern would be {{$W1}+scramble}.

### Static Text

Any text not contained within braces will not be modified for the final password. This allows you to include static as well as random elements in a password. Note that certain special characters such as { } [ ] ( ) \ + and | must be escaped to include them as static elements in the pattern. Below are examples of static text:

His name is {Word(MaleName)}

{number} \+ {number}

\({Word(noun)}\)

Note that if you wish to run a modifier against static text, you must enclose it within braces, but the text cannot be a valid function placeholder or modifier function. For example the pattern {dog house+swap} will return the phrase *hog douse*.

## Working with Password Templates

**Pattern Templates**

Pattern templates are located in the patterns.cfg file in Pafwert’s wordlist directory. The file is a plain text file, each line containing a comment or pattern. Lines that contain the # character in the first position are ignored. Each pattern must contain a name followed by a colon (:) followed by the pattern definition. See the patterns.cfg file for examples.

### Pattern Preferences

Pafwert randomly selects patterns from the patterns.cfg file. If you have one particular pattern that you want used more than others, simply place it in the pattern file multiple times, giving each copy a different name in the file.

### Using Sub Patterns

Because Pafwert parses patterns from the deepest level to the highest only checking a pattern’s syntax when it is actually processed, it is possible to perform some complex pattern replacements, including replacing function names, modifier names, or parameters. For example, consider the following valid patterns:

{word({noun|verb})}

{word(noun)+{ucase|lcase}}

{word(noun)+{u|l}case}

{row{1|2|3}}

### Modifiers as Functions

All modifiers can also be used as functions by including an additional parameter containing the characters to modify. For example, the following two patterns produce identical results:

{Replace({Word(noun)},e,3)}

{Word(noun)+replace(e,3)}

Sometimes it is more convenient to use the function form of a modifier but keep in mind that while they produce the same results, they are not functionally identical.

First, when using a modifier on a word, any back references will return the modified version of the word. Consider the following patterns:

{Word(noun)+scramble}

{Scramble({Word(noun)}

In the first pattern, $W1 will contain a scrambled noun, but in the second pattern $W1 will contain a noun and $W2 will contain the scrambled noun.

Another reason for using the function form of a modifier is because modifier parameters cannot contain back references. The pattern {word(noun)+replace($W1,e)} is not valid. In this case you would have to use the function form of the Replace modifier.

Finally, modifiers are processed much faster than functions, so only use the function form of a modifier when absolutely necessary.

**Pattern Errors**

Although the Pattern Designer application will syntax check your patterns and report possible errors, Pafwert itself will not perform these types of checks due to the performance overhead it requires. Consequently, Pafwert will try to use any pattern that you give it. If it cannot generate a password from a pattern, Pafwert will always try to return something, even if it is a random string of characters.

If you notice many random strings of characters, short or truncated passwords, or syntax from the patterns themselves appearing in Pafwert’s output, chances are you have a pattern error. You should always use the Pattern Designer application or the patterntester.exe command line tool to develop and test your patterns.

## Creating Strong Passwords

Strong password should meet the following criteria:

* They are at least ten characters, preferably more
* They make use of upper-case letters, lower-case letters, numbers, symbols, and spaces.
* They should be unique enough that they would never appear in a wordlist (hint: try googling the passwords to see how unique they are)
* They should be based on a strong random generator function
* They should mimic real-world patterns to aid retention
* They should utilize other memorization techniques

Pafwert patterns are designed to meet all of the above criteria. When designing patterns, try to keep them clean enough to easily be memorized but random enough to be effective. Remember that random characters and symbols are not the only way to make a strong password. A longer password of all lower-case letters is just as effective as a shorter password with numbers and symbols. A rule of thumb is that a two or three character increase in length is roughly equivalent to adding a number or symbol to a password. Of course, the most effective strategy is to use a variety of techniques in your patterns.

# Placeholder Functions

## Asc

**Syntax:** Asc(C*haracter*)

**Example Pattern:** {asc(a)}

**Example Return:** 97

Returns the ASCII value of any string *character*.

## Chr

**Syntax:** Chr(*ASCII Value*)

**Example Pattern:** {Chr(97)}

**Example Return:** a

Returns the ASCII character of a given *ASCII value*.

## Consonant

**Syntax:** Consonant([*Weight*])

**Example Pattern:** {Consonant}

**Example Return:** f

Returns a consonant from the English alphabet. If a positive *weight* is specified, this function will more likely return a common consonant. If a negative *weight* is specified, this function will more likely return a less common consonant.

## Entropy1, Entropy2, Entropy3

**Syntax:** Entropy1

**Example Pattern:** {Entropy1}

**Example Return:** 45

The three entropy functions return a random seed numbers based on time, computer specs, user input, or other factors. They provide an extended level of randomization and can be used to vary a pattern’s results for a particular user. For example, you could add an additional placeholder function based on the current entropy value. You should normally avoid using entropy functions directly, but keeping them as parameters to other functions or modifiers.

Entropy functions return a value between 0 and 99. Entropy1 is seeded with the current time and will return values slightly clustered around 50. Entropy2 is seeded with the current system uptime and will more likely return values closer to 100. Entropy3 is seeded with several system and application factors and will always be a totally random number between 1 and 100.

Note that these are the default entropy values and client application may provide for user-defined seeds that may not be consistent with these values.

## EndPunctuation

**Syntax:** EndPunctuation

**Example Pattern:** This is a sentence{EndPunctuation}

**Example Return:** This is a sentence?

This function returns one of the following characters: period (.), exclamation (!), question mark (?), or ellipses (…). Note that this function is weighted to more likely return a period or question mark.

## Keyboard

**Syntax:** Keyboard

**Example Pattern:** Hit the {Keyboard} key

**Example Return:** Hit the < key

This function returns a single character (letter, number, or symbol) that can be found on a standard English keyboard.

## LeftHand

**Syntax:** LeftHand

**Example Pattern:** With your left hand hit ‘{LeftHand}’

**Example Return:** With your left hand hit ‘s’

This function returns a single letter that is typed with the left hand on a standard English keyboard.

## Letter

**Syntax:** Letter([*Weight*])

**Example Pattern:** Box {Letter}

**Example Return:** Box g

Returns a letter from the English alphabet. If a positive weight is specified, this function will more likely return a common letter. If a negative weight is specified, this function will more likely return a less common letter.

## LongDay

**Syntax:** LongDay

**Example Pattern:** Next {longday}

**Example Return:** Next Thursday

Returns the full English name of a random day of the week.

## LongMonth

**Syntax:** {LongMonth}

**Example Pattern:** {longmonth} 15th

**Example Return:** November 15th

Returns the full English name of a random month.

## Now

**Syntax:** Now

**Example Pattern:** {Now}

**Example Return:** 12/28/2004 6:58:29 PM

Returns the current time. The result can be formatted with the Format modifier or function.

## Number

**Syntax:** Number([*Maximum*], [*Minimum*], [*Weight*], [*Decimal Places*])

**Example Pattern:** {Number(100)}

**Example Return:** 49

Returns a random number between *minimum* (default is 0) and *maximum* (default is 9). A positive weight will more likely select a number closer to *maximum*. A negative weight will more likely select a number closer to *minimum*. The returned number will be rounded to the nearest *decimal places*.

## NumberCode

**Syntax:** {NumberCode}

**Example Pattern:** {Word(Noun)} {NumberCode}

**Example Return:** shelf 3-34

Returns a random patterned number that may include repetition and one of the following symbols: - . , \ : to make it easier to remember.

Below are example number codes returned by this function:

|  |  |  |
| --- | --- | --- |
| (22-)8 | 0623 | 1712 |
| 272-67 | 9.1.5 | (21/4)6 |
| 8:6 | 77\85 | 4972 |
| 62-59 | 7/673 | 2626 |

## NumberPattern

**Syntax:** NumberPattern([*Length*])

**Example Pattern:** {NumberPattern(4)}

**Example Return:** 3130

Returns a random number, but produces a number that contains repeating elements. The *length* specifies the length in digits. For each digit, there is a chance that it will choose a number related to a previous digit. Below are example results from this function:

|  |  |  |
| --- | --- | --- |
| 15000 | 84868 | 88003 |
| 04567 | 99329 | 07077 |

## NumRow, NumRowFull

**Syntax:** NumRow

**Example Pattern:** {NumRow}

**Example Return:** 7

Returns a number from the numbers row of a standard English keyboard. NumRowFull returns either a number or a symbol from that same row. The character sets are as follows:

NumRow: 1 2 3 4 5 6 7 8 9 0

NumRowFull: 1 2 3 4 5 6 7 8 9 0 ` ~ ! @ # $ % ^ & \* ( ) \_ - + =

## Ordinal

**Syntax:** Ordinal(*Number*)

**Example Pattern:** Today is the {Ordinal(8)}

**Example Return:** Today is the 8th

This function returns the ordinal value (1st, 2nd, 3rd, etc.) of a number.

## Phonetic

**Syntax:** Phonetic(*String,* [*Style*])

**Example Pattern:** {Phonetic(b, 1)}

**Example Return:** Bravo

Returns the phonetic form of the S*tring* based on the selected style (1 or 2). The returned value is based on those listed in

Table 3.1: Phonetic return values

|  |  |  |
| --- | --- | --- |
| **Letter** | **Style 1** | **Style 2** |
| **A** | Alpha | Adam |
| **B** | Bravo | Baker |
| **C** | Charlie | Charles |
| **D** | Delta | David |
| **E** | Echo | Edward |
| **F** | Foxtrot | Frank |
| **G** | Golf | George |
| **H** | Hotel | Henry |
| **I** | India | Ida |
| **J** | Juliette | John |
| **K** | Kilo | King |
| **L** | Lima | Lincoln |
| **M** | Mike | Mary |
| **N** | November | Nora |
| **O** | Oscar | Ocean |
| **P** | Papa | Paul |
| **Q** | Quebec | Queen |
| **R** | Romeo | Robert |
| **S** | Sierra | Sam |
| **T** | Tango | Tom |
| **U** | Uniform | Union |
| **V** | Victor | Victor |
| **W** | Whiskey | Wililiam |
| **X** | X-Ray | X-Ray |
| **Y** | Yankee | Young |
| **Z** | Zulu | Zebra |

## Pronounceable

**Syntax:** Pronounceable

**Example Pattern:** {Pronounceable}

**Example Return:** axeda

Returns a word that somewhat resembles English, but is made up of random English digraphs and trigraphs. Below are some example results:

|  |  |  |
| --- | --- | --- |
| Oustless | Moflyst | Eastism |
| Chorthsome | Tiety | Astedness |
| Menible | Hasify | Voorly |

Note that because the algorithm is designed to make words that look like English, it may very well produce valid English words.

## RightHand

**Syntax:** RightHand

**Example Pattern:** {RightHand}

**Example Return:** p

This function returns a single letter that is typed with the right hand on a standard English keyboard.

## Row1, Row1Full, Row2, Row2Full, Row3, Row3Full

**Syntax:** Row1

**Example Pattern:** {Row1}

**Example Return:** r

Returns a number from the specified row of a standard English keyboard. Row1, Row2, and Row3 return a letter from the specified row, whereas Row1Full, Row2Full, and Row3Full return both letters and symbols.

## Sequence

**Syntax:** Sequence([*Length*])

**Example Pattern:** {Sequence(4)}

**Example Return:** qwer

This function returns a random sequence or pattern of letters and numbers based on their relative keyboard positions. Characters may be adjacent, mirrored, all one hand, all one row, etc.

## ShortDay

**Syntax:** {ShortDay}

**Example Pattern:** Next {shortday}

**Example Return:** Next Fri

Returns the abbreviated English name of a random day of the week.

## ShortMonth

**Syntax:** {ShortMonth}

**Example Pattern:** {shortmonth} 15th

**Example Return:** Nov 15th

Returns the abbreviated name of a random month.

## Smiley

**Syntax:** Smiley

**Example Pattern:** {Smiley}

**Example Return:** :-)

Returns classic ASCII smiley characters sequences.

## Space

**Syntax:** Space

**Example Pattern:** {Space}

**Example Return:**

Returns a single space. Useful for patterns where it is hard to embed a space due to the trimming functions of the generator.

## Symbol

**Syntax:** Symbol

**Example Pattern:** {Symbol}

**Example Return:** %

Returns any keyboard character that is not a number, letter, or space.

## Vowel

**Syntax:** Vowel

**Example Pattern:** {Vowel}

**Example Return:** e

Returns a vowel from the English alphabet. If a positive *weight* is specified, this function will more likely return a common vowel. If a negative *weight* is specified, this function will more likely return a less common vowel.

## Word(text file)

**Syntax:** Word(*filename*)

**Example Pattern:** {Word(noun)}

**Example Return:** corn

Returns a single random word or line from the specified text file. Text files should have one word or phrase per line.

# Word Modifiers

## A

**Syntax:** a

**Example Pattern:** {Word(Shape)+a}

**Example Return:** an oval

**Example Return:** a square

an apple

This modifier adds either ‘a’ or ‘an’ before the word, depending on whether it start with a vowel or not.

## Bracket

**Syntax:** +bracket

**Example Pattern:** {Word(noun)+bracket}

**Example Return:** <propane>

This modifier uses bracket pairs to make increase the number of character sets used in a password. Below are examples of the bracket modifier:

|  |  |  |
| --- | --- | --- |
| [*word*] | <*word* > | (*word* ) |
| [*word*] | |*word*| | \*word*/ |
| \**word*\* | [*word*] | {*word*} |
| /*word*/ | \*word*/ | /*word*\ |
| \*word*\ | <-*word*-> | ->*word*<- |

## Format

**Syntax:** +Format(*format string*)

**Example Pattern:** {Number(9999)+Format(#,#00.00)}

**Example Return:** 3,384.00

This modifier formats a word following the same syntax as the VB Format function (see http://msdn.microsoft.com/library/en-us/vbenlr98/html/vafctFormat.asp).

## Hide

**Syntax: +**Hide

**Example Pattern:** {{word(noun)}+hide}{$W1+left(2)}-{$W1+left(2))-{$W1}

**Example Return:** fi-fi-finger

This modifier hides a word so that you can refer to it via back references without actually showing the word.

## LCase

**Syntax:** +lcase

**Example Pattern:** {APPLE+lowercase}

**Example Return:** apple

Converts the word to all lower case letters.

## Left

**Syntax:** +left(*characters*)

**Example Pattern:** {Word(noun)+left(3)}

**Example Return:** ste

This modifier trims the word to the specified number of *characters*.

## Mid

**Syntax: +**Mid(*start, length*)

**Example Pattern:** {rocket+mid(2,3)}

**Example Return:** ock

Returns the portion of a word beginning at position *start* for the number of characters specified by *length*.

## Num2Word

**Syntax: +**Num2Word

**Example Pattern:** {Number(99)+Num2Word}

**Example Return:** Eighty-two

## Obscure

**Syntax:** +Obscure

**Example Pattern:** {Word(noun)+obscure}

**Example Return:** conthequence

This modifier increases password randomness, complexity and character sets by selecting from among approximately 100 character replacements. The purpose is to increase the number of possible password and strengthening a particular pattern. Below are some examples of this modifier:

|  |  |  |
| --- | --- | --- |
| windst0rm | embuhssy | inthtrumentation |
| topma$t | crozzroad | con'ainer |
| tr@nscription | aft=rnoon | plant3r |
| simuluhtion | estabrishment | $tarling |
| bure'u |  |  |

Note that the number of modifications is completely random, resulting in some words that are very obscured and others that are not modified at all.

The Obscure modifier performs zero, one or more of the following replacements:

|  |  |  |
| --- | --- | --- |
| ate with 8 | for with 4 | e with 3 |
| l with 1 | s with z | o with 0 |
| a with @ | s with $ | l with | |
| ait with 8 | a with | e with |
| ou with u | cc with x | oo with ew |
| and with & | are with r | ks with x |
| f with ph | ph with f | won with 1 |
| l with r | ee with eee | 000 with k |
| er with r | ex with x | ecs with x |
| m with mm | cke with x0 | qu with kw |
| a with ' | u with ' | ei with ee |
| one with own | oi with oy | om with um |
| a with aa | ew with u | us with is |
| y with ee | sh with ch | to with 2 |
| s with th | ck with q | ci with si |
| ie with iye | tion with shun | r with w |
| come with cum | cks with x | ight with ite |
| ing with 'n | th with f | tion with shun |
| too with 2 | why with y | won with 1 |
| your with yor | sc with sh | sh with th |
| ly with lee | er with uh | er with a |
| the with da | it is with | you with ya |
| l with w | th with d | a with u |
| th with ' | your with yer | ned with nt |
| e with \_ | t with + | e with = |
| can with kin | t with ' | ng with n' |
| red with hed | th with d | he with eh |
| h with | f with v | ha with o |
| v with f | v with b | N with |\| |
| ll with dd | ll with tt | dd with tt |
| h with ' | o with a | e with a |
| a with uh | a with u | oo with u |
| i with ih | a with ah | s with ss |
| t with tt | d with dd | at with @ |
| (Space) with (Blank) | with with w/ | t with d |
| t with dd | d with t | d with tt |
| cks with x | er with ah | (space) with a symbol |

## PigLatin

**Syntax:** +PigLatin

**Example Pattern:** www.{Word(Noun)+piglatin}.com

**Example Return:** www.egoyay.com

This modifier converts a word to Pig Latin.

## ProperCase

**Syntax:** +ProperCase

**Example Pattern:** {Word(MaleName)+propercase} {Word(Surname)+propercase}

**Example Return:** Rick Boise

This modifier capitalizes the first letter of each word.

## Quote

**Syntax:** +Quote

**Example Pattern:** {Word(Noun)+Quote}

**Example Return:** “guppy”

This wraps the placeholder in quotation marks.

## Random

**Syntax:** +Random

**Example Pattern:** {Word(Noun)+random}

**Example Return:** RACKET

This applies a randomly-selected modifier to the word.

## RandomCase

**Syntax:** +RandomCase

**Example Pattern:** {Word(Noun)}+RandomCase}

**Example Return:**

This modifier randomly capitalizes letters based on one of approximately 15 patterns such as all instances of a letter, two consecutive letters, all vowels, totally random, etc.

## Repeat

**Syntax:** +Repeat(*number*)

**Example Pattern:** {A+repeat(5)}

**Example Return:** AAAAA

This modifier repeats the word the specified number of times.

## Replace

**Syntax:** +Replace(*search, replace*)

**Example Pattern:** {Texas+replace(x, X)}

**Example Return:** TeXas

The modifier searches for all instances of the string specified by *search* and replaces them with the string specified by *replace*.

## Right

**Syntax:** +Right(*number*)

**Example Pattern:** (Brick+right(3)}

**Example Return:** ick

This modifier returns the rightmost characters of a word specified by *number*.

## RomanNumeral

**Syntax:** +RomanNumeral

**Example Pattern:** {Number(99)+romannumeral}

**Example Return:** XLVIII

This modifier converts a number to Roman numeral notation.

## Scramble

**Syntax:** +Scramble(*iterations*)

**Example Pattern:** {comedy+scramble(2)}

**Example Return:** dmoecy

This modifier swaps two letters in a word, one pair of letters swapped for each number of times specified by *iterations*.

## SentenceCase

**Syntax:** +SentenceCase

**Example Pattern:** {Word(Phrase)+SentenceCase}

**Example Return:** Walk the dog

This modifier capitalizes the first letter of a sentence, converting all other characters to lower case. Note that this function does not retain the capitalization of proper nouns or other capitalized words.

## Stutter

**Syntax:** +Stutter

**Example Pattern:**  {Word(noun)+Stutter}

**Example Return:** cocomodity

This modifier swaps repeats initial syllables in a word to create a stuttering effect.

## Swap

**Syntax:** +Swap

**Example Pattern:**  {dollar bill+swap}

**Example Return:** bollar dill

This modifier swaps the first letters of the first two words in the phrase modified.

## Trim

**Syntax:** +Trim

**Example Pattern:** {{Word(noun)} +trim}{$W1}

**Example Return:** dominopeanut

This modifier trims all leading and trailing spaces from the word.

## UCase

**Syntax:** +UCase

**Example Pattern:** {Word(noun)+ucase}

**Example Return:** RIVER

This modifier converts the entire word to uppercase characters.

# Pattern Examples

**Pattern**:

{Word(MaleName)}@{Word(bodypart)}.{com|net|org}

**Example Passwords:**

eldon@shoulder.org

parnell@elbow.net

rusty@neck.net

teodoro@chest.org

kipp@shoulder.net

christoph@nose.net

reece@arm.net

dario@abdomen.org

rorke@mouth.org

fin@neck.org

**Pattern**:

{Word(adjective)+ucase} {Word(noun)+obscure}

**Example Passwords:**

WORLDWIDE ttisp\_nsation

INEXPENSIVE tranquillity

DIVISIVE rotor

OBSTINATE stupidity

EXCITABLE trunk

NERVY tranquihllihty

WORLDWIDE mi$giving

ABLE-BODIED stink=r

RIGID swelling

GRAPHIC dispensation

**Pattern**:

{Word(adjective)+ucase} {Word(noun)+obscure}

**Example Passwords:**

california quail cakE

harpy eagle ANCHOVIEs

bowhead whale oysteRs

artiodactyls PEPPERs

woodland caribou dates

albertosaurus Fritters

brown pelican piEs

three-toed sloth ELK

attwater's prairie chicken Corn

redbilled oxpecker MUFFINS

**Pattern**:

{Mr.|Mrs|Ms.} {Word(Color)+ProperCase} the {Ordinal({Number})}

**Example Passwords:**

Mr. Rose the 5th

Mrs Timberwolf the 1st

Ms. Slate Gray the 5th

Ms. Navajo White the 8th

Mr. Fuchsia the 4th

Ms. Timberwolf the 2nd

Ms. Sea Green the 1st

Ms. Hazel the 2nd

Mrs Burgundy the 6th

Mrs Beige the 8th

**Pattern**:

{Word(EnglishCommon)}-{$W1}

**Example Passwords:**

snoop-snoop

gallium-gallium

accustoms-accustoms

witnesses-witnesses

telescopic-telescopic

sleeves-sleeves

atonality-atonality

sheathe-sheathe

duckbill-duckbill

quicksteps-quicksteps