Snopf Password Creation Algorithm

Hajo Kortmann

June 22, 2020

Snopf uses a deterministic algorithm for generating unique and strong passwords for every user login. The only crypto primitive used for generating passwords is SHA256 hashing, every password is essentially a SHA256 hash of the concatenation of the 256 bit secret stored on the Snopf device and a 128 bit request message which is unique for each password request.

Some services have different rules for valid passwords, for example the password must include a special character or a number. Passwords are iteratively generated until the created password meets the specific requirements for a given password request.

Variables

We introduce the short names S for the 32 byte secret on the Snopf and W for the final result of the password creation algorithm. W is a sequence of printable characters.

Data sent by the Host

For every password request the host must send the following data to the device:

1. M, the unique request message, a sequence of 16 bytes
2. l, an integer defining the length of the generated password
3. R, a set of the character inclusion rules (each rule is a set of characters)
4. r_{rep}, boolean variable; if True the password must not include repeated characters (e.g. 'aa')
5. r_{seq}, boolean variable; if True the password must not include sequences of characters (e.g. 'abc')
6. K, the keymap for the request, a sequence of integers of length 64

Request Message The request message must have a length of 16 bytes. It is defined as

$$M := \text{first 16 bytes of SHA256}\,(\text{service} \,||\, \text{account} \,||\, \text{master\_key} \,||\, \text{iteration})$$

Here *service* is the name of the service we want to log in to, *account* is our username or other account identifier for the service. The *master_key* is derived (using PBKDF2 with SHA256) from a master password we use for all passwords (might also be empty if the user doesn't want to use a

master password[1]). The *iteration* field is a number for each unique $(service, account)$ combination that is incremented every time a new password has to be created for a service, for example after a data breach at that service. Both fields for *service* and *account* are UTF-8 encoded strings without any terminating character like newline or null. *iteration* is an ASCII string representing the iteration integer and *master_key* is a 16 byte sequence.

Password Length The integer $l \in \{6, 7, .., 42\}$ is the length of the generated password. As we use 6 bit for each character and the SHA256 hash is 256 bits long we get the upper limit of 42 characters for the password. The reasoning for the lower boundary is stated below in the section 'Probability of Algorithm Convergence'.

Character Inclusion Ruleset As many services demand passwords to include characters of certain character groups the ruleset R is submitted. The ruleset R may include any of the following sets of characters which define which characters must be included:

1. The password must include a lowercase character, $R_{\text{lc}} = \{$'a', 'b', .., 'r'$\}$
2. The password must include an uppercase character, $R_{\text{uc}} = \{$'A', 'B', .., 'R'$\}$
3. The password must include a numerical character, $R_{\text{num}} = \{$'0', '1', .., '9'$\}$
4. The password must include a special character, $R_{\text{sc}} = \{$'-', '@', .., '>'$\}$

These character sets are later used to test whether the password includes any element of the submitted rules.

a (00)	b (01)	c (02)	d (03)	e (04)	f (05)	g (06)	h (07)
i (08)	j (09)	k (10)	l (11)	m (12)	n (13)	o (14)	p (15)
q (16)	r (17)	A (18)	B (19)	C (20)	D (21)	E (22)	F (23)
G (24)	H (25)	I (26)	J (27)	K (28)	L (29)	M (30)	N (31)
O (32)	P (33)	Q (34)	R (35)	0 (36)	1 (37)	2 (38)	3 (39)
4 (40)	5 (41)	6 (42)	7 (43)	8 (44)	9 (45)	- (46)	@ (47)
! (48)	? (49)	$ (50)	* (51)	+ (52)	= (53)	& (54)	# (55)
[(56)	] (57)	((58)	) (59)	. (60)	: (61)	< (62)	> (63)

Figure 1: Snopf Character Set

Keymap The first step of processing the SHA256 output is the Base64[2] transformation of the hash digest into a sequence of integers in the range $\{0, 1, .., 63\}$. We could directly map this result

[1]Note that *master password* and *master secret* are two independent variables. The master secret is the 256 bit secret stored on the Snopf device while the (optional) master password is a user input.

[2]Here we do not use the standard Base64 transformation definition where the output is a sequence of characters. The Base64 transformation in this context is the transformation of an input bit stream into a sequence of integers in the range $\{0, 1, .., 63\}$ using 6-bit chunks of the input stream.

to a sequence of characters using the character table C shown in figure 1 but to allow the definite exclusion of characters from the output sequence we use the keymap K. Using K we simply map the k-th integer i from the Base64 sequence to the k-th character in W:

$$W_k = C(K(i))$$

We require $0 \leq K(i) \leq 63$ for all $0 \leq i \leq 63$.

While C and the Base64 transformation are both bijective this often won't be the case for keymaps except the standard keymap where $K(i) = i$. For example, if we wanted to exclude all characters except lowercase letters the keymap would be defined as

$$K_{\text{lc}}(i) = i \bmod 18$$

and for only numerical characters as

$$K_{\text{num}}(i) = (i \bmod 10) + 36$$

and so on. The keymap chosen thus influences the entropy per character of W, see below at 'Password Entropy'.

Keymap validity for given rules The password creation rules defined above limit the creation of valid keymaps. If r_{rep} is selected we obviously must have more than one value in K. To be able to avoid sequences we must either have a keymap with only one entry or more than two. For simplicity a keymap is defined as valid if it includes more than two different values.

Each value of K belongs to one of the four character groups (lowercase, uppercase, numerical, special character). The size of the subset of the image of K must be at least ten for every character group where a must-inclusion-rule is selected. This limit is needed to guarantee convergence of the iterative algorithm, see below in 'Probability of Algorithm Convergence'.

Iterative Password Algorithm

To create a valid password for a given request we use an iterative algorithm where we manipulate the first byte of the request message as often as needed until the SHA256 hash output will produce a valid password according to the selected rules.

First, we define the function *replace_characters* which, according to the submitted rules, replaces repetitions or sequences of characters. Note that this function only converges given the boundary conditions above about the size of the keymap.

Further, we define the function *password_valid(W, R)* which returns True if the password is valid according to the inclusion rules in R. A valid password includes at least one element of each set in R.

The iterative password generation algorithm assumes that all passed parameters guarantee the convergence of the algorithm in less than 256 iterations. We treat the request message M as an integer sequence $M = (m_0, m_1, .., m_{15})$. To find a password we manipulate m_0 until the SHA256 hash results in a valid password according to the rules. The probability of the convergence of this algorithm is calculated in the following section.

Algorithm 1: Repetition and sequence replacement

function replace_characters;
Input : Integer sequence B of Length l, Keymap K, Boolean variables $r_{\text{rep}}, r_{\text{seq}}$
Output: Integer sequence B' where every $C(K(B'_i)) \neq C(K(B'_{i-1}))$ if $r_{\text{rep}} = \text{True}$ and every $C(K(B'_i)) \neq C(K(B'_{i-1}) + 1)$ if $r_{\text{seq}} = \text{True}$
$B'_0 := B_0$
foreach $B_i,\ 1 \leq i < l$ **do**
  $B'_i := B_i$
  while $[r_{\text{rep}} \wedge (C(K(B'_i)) = C(K(B'_{i-1})))] \vee [r_{\text{seq}} \wedge (C(K(B'_i)) = C(K(B'_{i-1})) + 1)]$ **do**
    $B_i := (B_i + 1) \bmod 64$
  end
end

Algorithm 2: Password valid function

function password_valid;
Input : Password W of length l, Inclusion ruleset R
Output: True if password is valid according to ruleset R, else False
$W_s = \{W_i \mid 0 \leq i < l\}$
for r **in** R **do**
  if $C \cap r = \emptyset$ **then**
    return False
  end
end
return True

Algorithm 3: Snopf password creation algorithm

Input : Secret S, Request Message M, Password Length l, Inclusion Rule set R, Repetition rule r_{rep}, Sequence rule r_{seq}, Keymap K
Output: Character Sequence (Password) W of Length l
for *1* **to** *256* **do**
  $B_{64} :=$ Base64(SHA256($S \mid\mid M$))
  replace_characters($B_{64},\ r_{\text{rep}},\ r_{\text{seq}},\ K$)
  $W := C(K(B_{64}))$
  if *password_valid(*$W,\ R$*)* **then**
    return W
  end
  $m_0 = (m_0 + 1) \bmod 256$
end
return *Failure*

Probability of Algorithm Convergence

There is a – for the chosen boundary conditions only theoretical – chance for the password creation algorithm to fail or rather need an unreasonable number of iterations to find a valid password. We assume the process of finding a password to be following a geometric distribution since repeated SHA256 hashing of the manipulated data is assumed to be a memoryless process with uniform random output in this context. The probability p of a success at a single attempt is calculated from the experimentally found mean of needed attempts μ:

$$p = 1/\mu$$

The probability of needing $X \leq n$ attempts for a success is then calculated by:

$$P(X \leq n) = 1 - (1 - p)^n$$

To find μ, 100,000 passwords for each password length setting were created; with different random inputs for S and M at each try.

To test the worst case scenario all character inclusion rules were used for testing algorithm convergence ($R = \{R_{\text{lc}}, R_{\text{uc}}, R_{\text{num}}, R_{\text{sc}}\}$). Also an unbalanced keymap was used, which included just 10 (minimum number) entries for three of the rules, thus lowering the chance for a certain character to be included in the output.

In table 1 we can see that there is only a theoretical chance for the algorithm to fail given the shortest possible password of length $l = 6$. For $l > 6$ we rarely need more than 2 tries to find a valid password.

Password length	**mean**	$1 - P(k = 1)$	$1 - P(k \leq 10)$	$1 - P(k \leq 255)$
6	4.8	0.7922	0.0974	0.0
10	1.9	0.4748	0.5816e-3	0.0
20	1.1	0.0951	0.606e-12	0.0

Table 1: Probability of algorithm convergence

Password Entropy

The entropy of a password generated by Snopf depends on the length, chosen keymap and the selected rules. A password generated using the default settings (uniformly distributed keymap, no rules) results in a entropy of more than 128 bits for a password with a length of 22 characters. An entropy of 128 bit is deemed to be very safe for most passwords and 22 characters is short enough to be practical therefore the default password length setting is set to 22 characters. Whenever a a non-uniform keymap is used, the entropy per character is reduced and raising the password length might be advised.

Given the relative frequency $x(k)$ for every $k \in K_I = \{K(i) \mid i \in \{0, 1, ..63\}\}$ the informational entropy in number of bits for a single character is given by

$$H_c = -\sum_{k \in K_I} x(k) \log_2 x(k)$$

and the entropy for a password is then obviously given by $H = l \cdot H_c$.

Table 2 shows the entropy for some lengths and often used keymaps without extra rules.

When password creation rules are enforced the resulting entropy will often be reduced. A solution for calculating the resulting entropy for any given keymap and a non empty rule set has not been found yet by the author. Snopf shows a warning in the GUI whenever rules are active, that the shown password entropy is a rough guess and will be lower in reality.

Keymap	**Keymap set size**	H **for** $l = 1$	H **for** $l = 6$	H **for** $l = 22$	H **for** $l = 42$
All	64	6.0	36	132	252.0
Alphanumerical	46	5.44	32.63	119.63	228.38
Letters	36	5.14	30.75	112.75	215.25
Lowercase Uppercase only	18	4.16	24.93	91.42	174.54
Hex	16	4.0	24.0	88.0	168.0
Numerical	10	3.32	19.91	72.99	139.35

Table 2: Password entropy for different keymaps