Certified Password Quality

A Case Study Using Coq and Linux Pluggable Authentication Modules

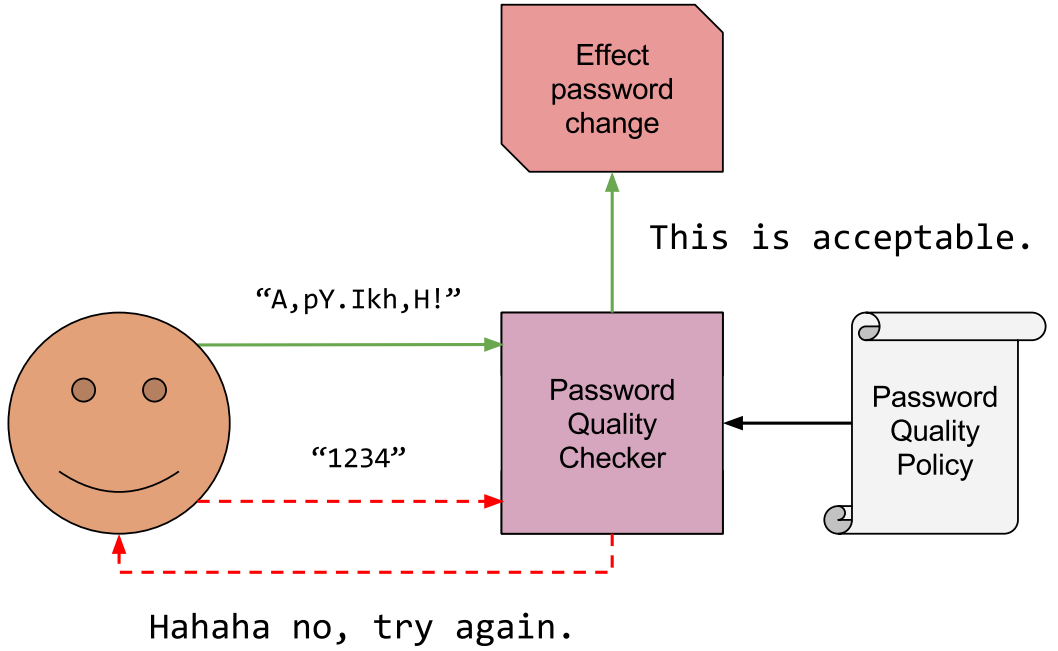
João F. Ferreira[1, 2], **Saul A. Johnson[1]**, Alexandra Mendes[1] and Philip J. Brooke[1]

[1] School of Computing, Media and the Arts, Teesside University

[2] HASLab/INESC TEC, Universidade do Minho

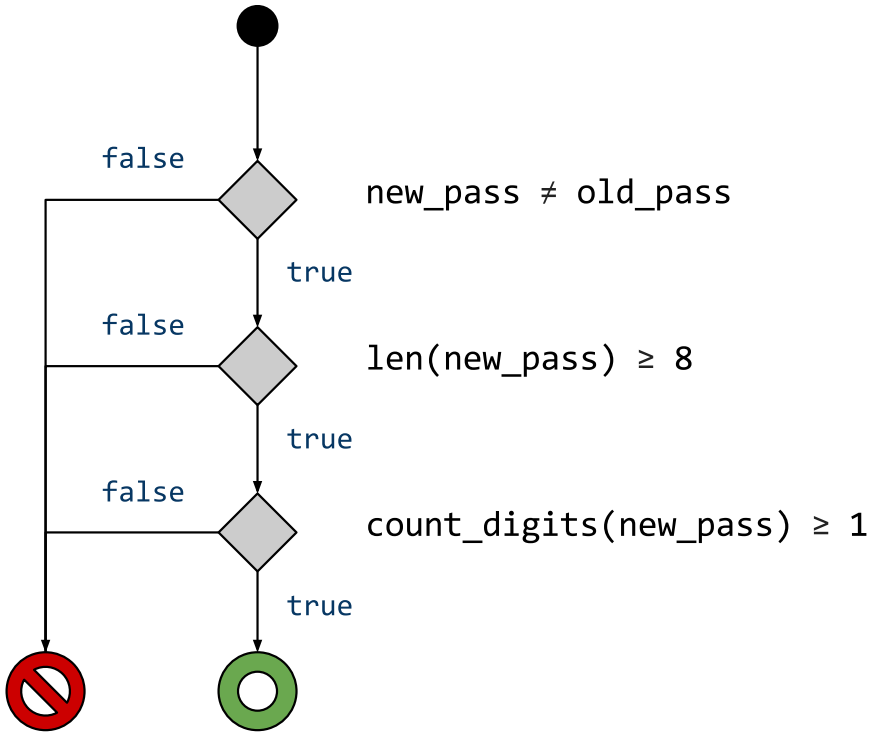
The Importance of Strong Passwords

- It is well-established that without an enforced password policy, users tend to create poor passwords.
- If an attacker authenticates as a legitimate user by guessing their password, all security measures deployed to keep unauthenticated users out of the system are rendered irrelevant. **All bets are off.**



The Shape of a Password Quality Policy

- The subset of passwords a system will accept is dictated by its *password quality policy,* which is enforced by its *password quality checker*.
- For our purposes, an acceptable password is whatever the system administrator configures it to be.
- We want to offer the sysadmin assurance that the password quality checker will correctly enforce the policy they create.



Password Quality Checkers Today

- Focusing on Linux systems, two largely identical pluggable authentication modules, *pam_cracklib* and *pam_pwquality*, are widely deployed (on millions of machines) to prevent users from creating unacceptably weak passwords.
- The *passwd* tool on most systems will use one of these modules for password quality checking during password creation/change.
- Both modules (pam_cracklib, in particular – 1996) are dated, and use virtually identical code for the actual password quality checking.

So let's ask ourselves…

Should we trust this code to correctly enforce the password quality policy as specified by the system administrator?

The *pam_cracklib* Default Password Policy

Passwords must:

- Not be identical to the previous password, if any.
- Not be palindromes.
- Not be a rotated version of the old password, if any.
- Not contain case changes only in relation to the previous password, if any.
- Have a Levenshtein distance of 5 or greater from the previous password, if any (difok=5).
- Be at least 9 characters long (minlen=9), however:
  - Passwords may be 1 character shorter if they contain at least 1 lower case letter (lcredit=1).
  - Passwords may be 1 character shorter if they contain at least 1 upper case letter (ucredit=1).
  - Passwords may be 1 character shorter if they contain at least 1 digit character (dcredit=1).
  - Passwords may be 1 character shorter if they contain at least 1 other character (ocredit=1).
  - This shortening of minimum length will stack, making for a minimum length of 9 - 4 = 5 for passwords containing all 4 classes.
  - Effective minimum length is, then M = m − c where M is the effective minimum length, m is the configured minimum length and c is the number of character classes present in the string.

C Code Used by Linux-PAM

- We identified the functions within *pam_cracklib* necessary to implement its **default** policy.
- Some of these were already nicely separated in to their own methods.
- Others were not so cleanly separated – for example, comparison of the new password to the old password, which was just a call to `strcmp`.

```
static int palindrome(const char *new)
{
  int i, j;
  i = strlen (new);
  for (j = 0; j < i; j++)
    if (new[i - j - 1] != new[j])
      return 0;
  return 1;
}
```

Implementation and Specification

- **First we implemented:** using the module documentation and original source code from the official repository, we implemented these checkers in Gallina, the implementation language of Coq.
- **Then we specified:** by identifying and writing proofs of various desirable properties for our checkers. Identifying these desirable properties made up a significant chunk of work.

```
Lemma string_reverse_involutive
  : ∀ (s : string),
  string_reverse (string_reverse s) = s.
Proof.
  induction s as [| c s'].
  (* Base case *)
  - simpl. reflexivity.
  (* Inductive step *)
  - simpl.
    rewrite (string_reverse_unit
          (string_reverse s') c).
    rewrite IHs'.
    auto.
Qed.
```

Executable Specifications

```
Definition palindrome (s : string) : bool :=
    s ==_s (string_reverse s).
```

We are able to express some checker functions as executable specifications. In this example, we at once define what it means for a string to be a palindrome while also defining a function for checking whether or not a given string is a palindrome.

Specification by Theorem

```
Theorem prefix_correct : ∀ (s1 s2 : string),
    prefix s1 s2 = true ⇔
    substring 0 (length s1) s2 = s1.
```

For other functions, we could capture their correct behaviour in a theorem which serves as their proof of correctness. Here, for instance, we define what we mean when we say that one string is a prefix of another. The `substring` and `length` functions used here are in Coq's standard library, with accompanying proofs.

Specification by Property

```
Lemma hamming_distance_zero_for_identical
    : ∀ (s : string),
    hamming_distance s s = Some 0.

Lemma hamming_distance_defined_for_same_length
    : ∀ (a b : string),
    length a = length b → hamming_distance a b ≠ None.
```

Sometimes we might only be interested in increasing our confidence our code by proving various desirable properties about it. Here, for example, we create a lemma which states that identical strings have a Hamming distance of 0 from each other and for any two strings of the same length, Hamming distance is defined.

Pulling Our Checkers Together into a Policy

Finally, we combined these into one password policy function with signature:

```
PasswordTransition → CheckerResult
```

Pulling Our Checkers Together into a Policy

Finally, we combined these into one password policy function with signature:

$$\texttt{PasswordTransition} \to \texttt{CheckerResult}$$

Equivalent to:

$$\underbrace{\texttt{option string}}_{\text{Old password}} \times \underbrace{\texttt{string}}_{\text{New password}} \to \underbrace{\texttt{option string}}_{\text{Error}}$$

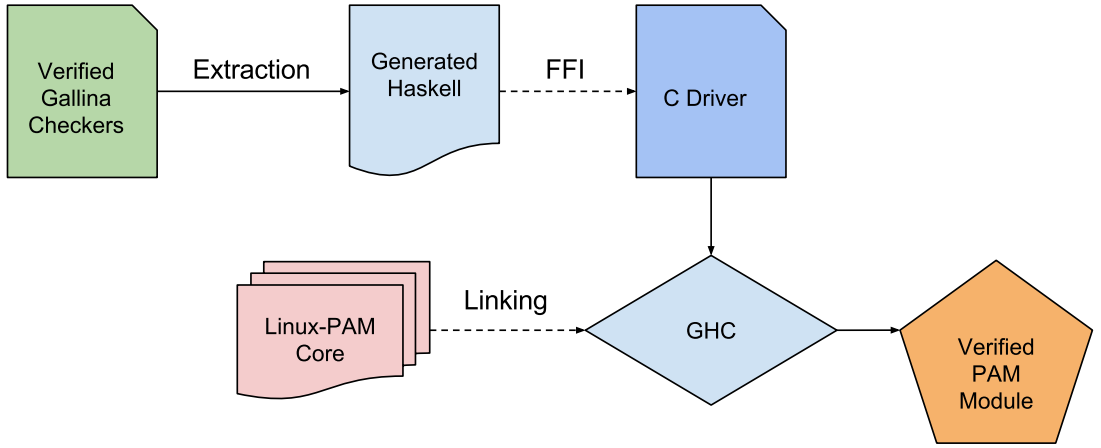
An Example of a Password Policy

```
Definition pwd_quality_policy := [
    diff_from_old_pwd ;
    not_palindrome ;
    not_rotated ;
    not_case_changes_only ;
    levenshtein_distance_gt 5 ;
    credits_length_check 8
].
```

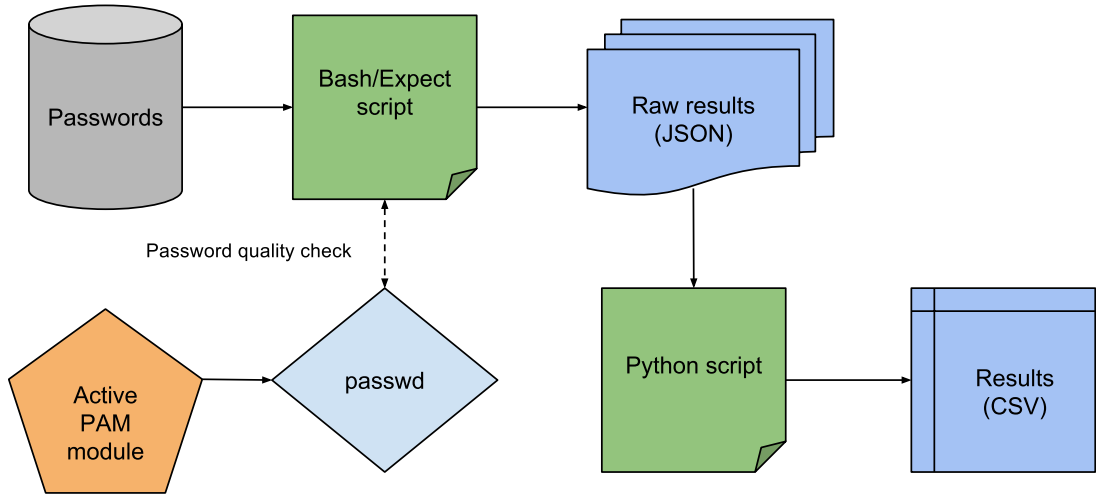
- This password policy encodes the default behaviour of *pam_cracklib* and *pam_pwquality*.
- The last one *credits_length_check* represents a seemingly home-grown algorithm we found in the original modules.
- We were surprised by how easily we were able to implement this custom functionality.

Building a PAM Module Using Verified Code

- It was anticipated that we would be able to extract the verified Gallina to Haskell, then use a C driver conforming to the PAM interface to call into it using the Haskell FFI.
- After compiling and linking this driver, we'd have a self-contained PAM module that runs verified code.



Evaluating Our Verified Module



- We located a database of 5 million leaked plaintext passwords from various sources (without accompanying usernames) and randomly sampled 100,000.
- We ran each of these through the system *passwd* executable configured with the original pam_cracklib module and our verified module in turn.

Default Policy: Original vs. Verified Modules

Number of passwords accepted by unverified, original module:

56574

Number of passwords accepted by verified module:

56574

A Research-Informed Policy

- We were very quickly able to reconfigure our checker according to research by Kelley et al. (2012)
- Their research suggests that the most effective countermeasure against password guessing attacks is enforcing long passwords (16 characters or greater).
- Under the very simple password policy on the right, **970** of our 100,000 passwords were accepted.

```
Definition pwd_quality_policy := [
    plain_length_check 16
].
```

One specific checker caught our attention…

For context, *pam_cracklib* treats uppercase letters, lowercase letters, digits and symbols as 4 separate character “classes”.

The configuration option *maxclassrepeat* restricts the number of characters of the same class that can appear consecutively in a password.

For example, when *maxclassrepeat* is set to 3…

“pwd1234”

Rejected – more than 3 digits in a row.

“password123”

Rejected – more than 3 lowercase letters in a row.

“PASsw0rd123”

Accepted!

We set it to 1…

And It Broke…

Unverified, original module, configured with default policy extended with maxclassrepeat=1:

56574

Verified module, same configuration:

371

Whence this Bug?

maxclassrepeat=N

Reject passwords which contain more than N consecutive characters of the same class. The default is 0 which means that this check is disabled.

```
if (opt->max_class_repeat > 1
 && sameclass > opt->max_class_repeat)
{
    return 1;
}
```

N.B. A bug report submitted to maintainers along with pull request containing a fix was accepted and merged.

Limitations and Caveats

- The compiled, verified PAM module is predictably a significantly larger file than the original (≈10 times smaller). This is likely to make it unappealing for use when storage space is tightly constrained.
- While not drastic, the execution time of the verified module is around 1.28 times that of the original. When behaviour of both is expected to be identical, how can we motivate adoption?
- The Haskell extractor built in to Coq is, as far as we are aware, not formally verified to be semantically transparent.

Future Work

We are currently developing a domain-specific language (DSL) for creating password quality checkers that are correct-by-construction. Here's a little preview of our progress so far:

```
Definition comprehensive8 :=
  (enforce new_pwd (min length 8) “Password too short!")
  /*\ (enforce new_pwd (min count_upper 1) “Must contain an uppercase letter!")
  /*\ (enforce new_pwd (min count_lower 1) “Must contain a lowercase letter!")
  /*\ (enforce new_pwd (min count_digit 1) “Must contain a digit!")
  /*\ (enforce new_pwd (min count_other 1) “Must contain a symbol!").
```

Future Work (contd.)

- We also hope to substantially reduce the size of the unverified C driver by stripping out functionality that is not absolutely necessary or that has been made redundant by our verification efforts.
- There is potential scope for future research involving verified compilers such as CertiCoq, to address the unverified Haskell extractor limitation.

Everything’s on GitHub!

https://github.com/sr-lab | @lambdacasserole