

Password Compromise Monitoring Tool

Adam Mirre

Master's thesis
2023



Tomas Bata University in Zlín
Faculty of Applied Informatics

Akademický rok: 2022/2023

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení: **Bc. Adam Mirre**
Osobní číslo: **A21819**
Studijní program: **N0613A140022 Informační technologie**
Specializace: **Kybernetická bezpečnost**
Forma studia: **Kombinovaná**
Téma práce: **Nástroj pro monitoring kompromitace hesel**
Téma práce anglicky: **Password Compromise Monitoring Tool**

Zásady pro vypracování

1. Specifikujte požadavky na systém s ohledem na jeho zabezpečení.
2. Vyberte vhodné zdroje dat pro ověření kompromitace hesel.
3. Navrhněte systém pro online správu vlastní databáze kompromitovaných loginů.
4. Navržený systém implementujte v testovacím prostředí a ověřte jeho funkčnost.
5. Ověřte izolaci uživatelských účtů Vašeho systému. Popište bezpečnostní mechanismy, které ji zajišťují.

Seznam doporučené literatury:

1. S. Stamm, B. Sterne, and G. Markham, Reining in the Web with Content Security Policy, in Proceedings of the 19th International Conference on World Wide Web, ser. WWW '10, 2010, p. 921–930
2. OWASP top 10 web application security risks. [Online]. Available: <https://owasp.org/www-project-top-ten/>
3. Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. 2016. CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16). Association for Computing Machinery, New York, NY, USA, 1376–1387. <https://doi.org/10.1145/2976749.2978363>
4. Jean-Philippe Aumasson, Serious Cryptography, No Starch Press, 2017, ISBN-13: 9781593278267
5. Sivathanu, Gopalan & Wright, Charles & Zadok, Erez. (2005). Ensuring data integrity in storage: techniques and applications. 26-36. 10.1145/1103780.1103784.
6. A. K. Pandey et al., "Key Issues in Healthcare Data Integrity: Analysis and Recommendations," in IEEE Access, vol. 8, pp. 40612-40628, 2020, doi: 10.1109/ACCESS.2020.2976687.
7. "Site Isolation – The Chromium Projects." [Online]. Available: <https://www.chromium.org/Home/chromium-security/site-isolation>

Vedoucí diplomové práce: **Ing. David Malaník, Ph.D.**
Ústav informatiky a umělé inteligence

Datum zadání diplomové práce: **28. července 2023**
Termín odevzdání diplomové práce: **25. srpna 2023**



doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 7. prosince 2022

I hereby declare that:

- I understand that by submitting my Master's thesis, I agree to the publication of my work according to Law No. 111/1998, Coll., On Universities and on changes and amendments to other acts (e.g. the Universities Act), as amended by subsequent legislation, without regard to the results of the defence of the thesis.
- I understand that my Master's Thesis will be stored electronically in the university information system and be made available for on-site inspection, and that a copy of the Master's Thesis will be stored in the Reference Library of the Faculty of Applied Informatics, Tomas Bata University in Zlín.
- I am aware of the fact that my Master's Thesis is fully covered by Act No. 121/2000 Coll. On Copyright, and Rights Related to Copyright, as amended by some other laws (e.g. the Copyright Act), as amended by subsequent legislation; and especially, by §35, Para. 3.
- I understand that, according to §60, Para. 1 of the Copyright Act, Tomas Bata University in Zlín has the right to conclude licensing agreements relating to the use of scholastic work within the full extent of §12, Para. 4, of the Copyright Act.
- I understand that, according to §60, Para. 2, and Para. 3, of the Copyright Act, I may use my work – Master's Thesis, or grant a license for its use, only if permitted by the licensing agreement concluded between myself and Tomas Bata University in Zlín with a view to the fact that Tomas Bata University in Zlín must be compensated for any reasonable contribution to covering such expenses/costs as invested by them in the creation of the thesis (up until the full actual amount) shall also be a subject of this licensing agreement.
- I understand that, should the elaboration of the Master's Thesis include the use of software provided by Tomas Bata University in Zlín or other such entities strictly for study and research purposes (i.e. only for non-commercial use), the results of my Master's Thesis cannot be used for commercial purposes.
- I understand that, if the output of my Master's Thesis is any software product(s), this/these shall equally be considered as part of the thesis, as well as any source codes, or files from which the project is composed. Not submitting any part of this/these component(s) may be a reason for the non-defence of my thesis.

I herewith declare that:

- I have worked on my thesis alone and duly cited any literature I have used. In the case of the publication of the results of my thesis, I shall be listed as co-author.
- The submitted version of the thesis and its electronic version uploaded to IS/STAG are both identical.

In Zlín; dated:

.....

Student's Signature

ABSTRAKT

Cielom diplomovej práce bolo vytvoriť a popísať nástroj na monitoring kompromitácie hesiel, ktorý umožní používateľovi overiť kompromitáciu. Teoretická časť pojednáva o kryptografických základoch a aplikácii hashov, venuje sa dejinám, použitiu a zmysluplnosti hesiel a skúma bezpečnostné mechanizmy, ktoré bežne využívajú webové prehliadače, ako napríklad Content Security Policy, a tiež spôsoby využitia týchto mechanizmov vo webových aplikáciách. Ďalej sa pojednáva o výbere vhodných zdrojov dát pre overenie kompromitácie, a časť končí rozborom scenárov nasadenia. Praktická časť vysvetľuje architektúru aplikácie vytvorenej k diplomovej práci, vybrané implementačné detaily, pridružené nástroje a metódy použité na overenie správnosti správania programu.

Kľúčová slova: Hesla, Monitoring kompromitace, Únik hesla, Bezpečnost webu

ABSTRACT

The aim of the thesis has been to program and describe a tool that enables the user to verify the potentiality of a credential breach. The theoretical part analyses the cryptographic foundations of hashing and their application, the history, usage as well as rationale of passwords. It explores security mechanisms commonly employed by web browsers, such as Content Security Policy, and also how web applications can make use of them. Potential data sources are then evaluated, and the part concludes by pondering deployment scenarios. The practical part explains the application architecture, chosen implementation details, associated tooling, and methods used to validate the correctness of program's behaviour.

Keywords: Passwords, Compromise monitoring, Password breach, Web security

With respect and thanks to my supervisor Dr. Malaník for all advice, and to M. for all the time and support, and to my family and friends for all their thoughts and encouragement.

It's all about perspective...

– Jones in Andy Andrews' *The Noticer*

TABLE OF CONTENTS

INTRODUCTION	10
I THEORETICAL PART	11
1 CRYPTOGRAPHY PRIMER	13
1.1 HASH FUNCTIONS	13
1.1.1 Types and use cases.....	13
1.1.2 Why are hashes interesting	14
1.2 TLS	15
2 PASSWORDS	16
2.1 PROGRAM-IMPOSED CONSTRAINTS	16
2.1.1 Short arbitrary length	17
2.1.2 Restricting special characters	17
2.1.3 Character composition requirements	18
2.1.4 Other common issues	19
3 WEB SECURITY	20
3.1 SITE ISOLATION.....	20
3.2 CROSS-SITE SCRIPTING	20
3.3 CONTENT SECURITY POLICY	21
4 CONFIGURATION	23
4.1 SAFETY CONSIDERATIONS	23
4.2 POSSIBLE ALTERNATIVES.....	24
5 COMPROMISE MONITORING	25
5.1 DATA SOURCES	26
5.1.1 Local Dataset Plugin	27
5.1.2 Have I Been Pwned? Integration	29
6 DEPLOYMENT RECOMMENDATIONS	30
6.1 TRANSPORT SECURITY.....	30
6.2 CONTAINERISATION	30
7 SUMMARY	32
II PRACTICAL PART	32
8 INTRODUCTION	34
8.1 KUDOS.....	34
9 DEVELOPMENT	35

9.1	COMMIT SIGNING.....	35
9.2	CONTINUOUS INTEGRATION	36
9.3	SOURCE CODE REPOSITORIES	38
9.4	TOOLCHAIN.....	38
10	APPLICATION ARCHITECTURE.....	40
10.1	PACKAGE STRUCTURE.....	41
10.1.1	Internal package.....	41
10.2	LOGGING	41
10.3	AUTHENTICATION	42
10.4	SQLI PREVENTION	42
10.5	CONFIGURABILITY	42
10.6	EMBEDDED ASSETS.....	44
10.7	COMPOSABILITY	44
10.8	SERVER-SIDE RENDERING	45
10.9	FRONTEND.....	45
10.9.1	Frontend experiments.....	46
10.10	USER ISOLATION.....	46
10.11	ZERO TRUST PRINCIPLE AND CONFIDENTIALITY	48
11	IMPLEMENTATION	49
11.1	DHALL CONFIGURATION SCHEMA.....	49
11.2	DATA INTEGRITY AND AUTHENTICITY.....	52
11.3	DATABASE SCHEMA.....	53
12	DEPLOYMENT.....	55
12.1	ROOTLESS PODMAN.....	55
12.1.1	Sanity checks	57
12.1.2	Database isolation from the host	58
12.1.3	Health checks.....	58
12.2	REVERSE PROXY CONFIGURATION	59
13	VALIDATION.....	60
13.1	UNIT TESTS	60
13.2	INTEGRATION TESTS	61
13.2.1	func TestUserExists(t *testing.T)	61
13.3	TEST ENVIRONMENT.....	63
13.3.1	Deployment validation	63

14 APPLICATION SCREENSHOTS	66
CONCLUSION	72
REFERENCES	73
LIST OF ABBREVIATIONS	78
LIST OF FIGURES	79
LIST OF TABLES	80
LIST OF LISTINGS.....	81
LIST OF APPENDICES	82
1.1 GIT SIGNING KEYS.....	83
1.2 ZSTD-COMPRESSED TARBALL OF THE PCMT SOURCE CODE REPOS- ITORY	83
1.3 ZSTD-COMPRESSED TARBALL OF THE PCMT CONFIGURATION SCHEMA REPOSITORY	83
1.4 ZSTD-COMPRESSED TARBALL OF THE L ^A T _E X SOURCES OF THIS THESIS	83
1.5 BLAKE3 CHECKSUMS.....	83
1.6 SHA3-256 CHECKSUMS.....	84
2.1 WHY GO	85
2.2 WHY NIX/DEVENV	85
3.1 LINUX.....	86
3.2 GNU/LINUX.....	86
3.3 THE PROGRAM	86

INTRODUCTION

Passwords. Everybody reading this text most assuredly recalls at least *some* of their own. The security-minded person perhaps even dozens. They are complex and at least twelve characters long. They are only ever used in the one place they were created for. And they are definitely getting rotated at least once a year. Or are they?

A token so ubiquitous that it becomes tiring for a human being to keep track of all the places where it is required in one form or another. At some point, it almost feels easier to stop caring and use the password intended for *the other site* for this one, too. What harm could that possibly do. The answer might well be “unimaginable”, depending on the service in question, its relevance to the person being discussed, and also on *how many other* services happen to share this password. Does a service require registration? No problem, the password will be the name of the cat (or dog, or the youngest child) plus current year, so as to make it more *secure*. It is the password rotation day again this month, a handful of logins will be disabled if their passwords are not changed in the next couple of hours. No worries, it is already covered by a combination of the current month and the name of the specific service for each of them. A neat system. But just in case it got forgotten and lost in the fragments of this hectic lifestyle, the passwords need to be written down on a sticker note. Not to worry, nobody knows, it is hidden under the keyboard, it is practically invisible.

These are all examples of poor password practices on user’s side; some might have been circumstantially helped to, such as the too frequent forced password rotation, others could be ascribed to users not being sufficiently well-versed in the intricacies of password hygiene.

Inevitably, these passwords are going to get appropriately treated, be in the form of misuse from a nosy colleague that finds the sticker note, or if the user account is ever a target of an attack, the password’s *only* role, to gate the access, will likely not stand much of a chance.

This thesis tangentially covers user-relating issues like the ones described above, but rather than attempting to remedy them with prevention, it mainly focuses on dealing with the acute consequence of such behaviour: a password breach.

The thesis consists of two parts: a theoretical one, which provides theoretical background to concepts and processes that are used in the so called *practical* part, where it is further described what exactly has been done and how.

In the theoretical part, password hash functions and hash cracking are mentioned, and within the browser context a special spotlight is given to Content Security Policy and Cross-site scripting. Program's configuration schema is conceived, the choices of local and online data sources are explained, and recommended deployment set-up is described.

The practical part discusses application architecture decisions, development process, implementation details and validation methods utilised when building the subject web application. The program does not have too many dependencies and is relatively lightweight, which means that anybody with even little experience should be able to potentially run their own private instance, if they so choose.

The purpose of the program is to allow users to learn if they were breached, and the application developed as an integral part of this thesis should enable them to quickly and privately check potential compromise status against configured local and online data sources. Of course, the quality of the compromise monitoring depends on access to quality data, which is partially in the purview of the application operator.

Breach data is not exactly a publicly traded commodity and can be relatively hard to make sense of, given that we are talking about literal *terrabbytes* of data available, if there is even a slightest interest to find it online. Breaches do happen, and can, of course, inform the decision to stay or leave the service, but there is not always a choice element involved, or only a limited amount. Either way, knowledge is light and as such precedes informed decision-making. Abstracting away the ugly parts and offering users an understandable interface might likely result in their improved security posture, if anything.

As per the security posture of the users, in order not to weaken it, the in-application administrative-level users should only be able to configure online and local data sources and initially set up user accounts but should *not* have access to users' search queries or credential entries. Sensitive user data should be encrypted at rest and not even administrative-level users should be able to read them.

Terminology is located in Appendix 3, feel free to give it a read.

The author has been striving to utilise modern tooling and development practices in an effort to build a maintainable and long-lasting piece of software that serves its users well. When deployed, it could provide a real value.

I. THEORETICAL PART

1 CRYPTOGRAPHY PRIMER

1.1 Hash functions

Hash functions are algorithms used to help with a number of things: integrity verification, password protection, digital signature, public-key encryption and others. Hashes are used in forensic analysis to prove authenticity of digital artifacts, to uniquely identify a change-set within revision-based source code management systems such as Git or Mercurial, to detect known-malicious software by anti-virus programs or by advanced filesystems in order to verify block integrity and enable repairs, and also in many other applications that each person using a modern computing device has come across, such as when connecting to a website protected by the famed HTTPS.

The popularity of hash functions stems from a common use case: the need to simplify reliably identifying a chunk of data. Of course, two chunks of data, two files, frames or packets could always be compared bit by bit, but that can get prohibitive from both cost and energy point of view relatively quickly, with transport channels being often insecure and unreliable. That is when the hash functions come in, since they are able to take a long input and produce a short output, named a digest or a hash value. The function also only works one way. A file, or any original input data for that matter, cannot be reconstructed from the hash digest alone by somehow *reversing* the hashing operation, since at the heart of any hash function there is essentially a compression function.

Most alluringly, hashes are frequently used with the intent of *protecting* passwords by making those unreadable, while still being able to verify that the user knows the password, therefore should be authorised. As the hashing operation is irreversible, once the one-way function produces a short a digest, there is no way to reconstruct the original message from it. That is, unless the input of the hash function is also known, in which case all it takes is hashing the supposed input and comparing the digest with existing digests that are known to be digests of passwords.

1.1.1 Types and use cases

Hash functions can be loosely categorised based on their intended cryptographic application to *password protection*, *integrity verification*, *message authentication* hashes. Each of them possesses unique characteristics and using the wrong type of hash function for the wrong job can potentially result in a security breach.

As a contrived example, suppose MD5, a popular hash function internally using the same data structure - *Merkle-Damgård* (MD) construction - as BLAKE3. The former produces 128 bit digests, compared to the default 256 bits of output and no upper ($< 2^{64}$ bytes) limit (Merkle tree extensibility) for the latter. Aside from MD5 considered to be *broken* in regard to collision resistance [1] [2] (and have theoretically weakened resistance to preimages [3] [4]), a list of differences could be mentioned; however, they both have one thing in common: they are *designed* to be *fast*. The latter cryptographic hash function, is conjectured to be *random oracle indistinguishable*, secure against length extension, and was built with pre-image and collision resistance in mind. That said, it is also in fact faster than all of MD5, SHA3-256, SHA-1 and even Blake2 family of functions [5].

```
m := x
m' := y
MD5(m) = MD5(m')
```

Listing 1.1 Broken collision resistance of MD5

However, the default use case of both MD5 and BLAKE3 (unkeyed) is to (quickly) verify integrity of a given chunk of data, not to secure a password by hashing it first, which poses a big issue when used to...secure passwords by hashing them first.

Password hashing functions such as `argon2` or `bcrypt` are good choices for *securely* storing passwords representations, namely because they place CPU and memory burden on the machine that is computing the digest. In case of the mentioned functions, *hardness* is even configurable to satisfy the greatest possible array of scenarios. These functions also forcefully limit potential parallelism, thereby restricting the scale at which exhaustive searches performed using tools like `Hashcat` or `John the Ripper` could be at all feasible. Additionally, both functions can automatically add random *salts* to passwords, automatically ensuring that no copies of the same password provided by different users end up hashing to the same digest value, which for practical purposes obviates large-scale old-school hash cracking [6], [7].

1.1.2 Why are hashes interesting

As already hinted, hashes are often used to store a *logical proof of the password*, rather than the password itself. Especially services storing hashed user passwords happen to non-voluntarily leak them. Using a wrong type of hash for password hashing, weak hash function parameters, reusing *salt* or the inadvertently *misusing* the hash function in some other way, is a sure way to spark a lot of interest [8], [9], [10].

Historically, plain-text passwords have also leaked enough times (or weak hashes have

been cracked) that anyone with enough interest had more than sufficient amount of time to additionally put together neat lists of hashes of the most commonly used passwords [11], [12], [13], [14].

So while a service might not be storing passwords in *plain text*, which is a good practice, using a hashing function not designed to protect passwords does not offer much additional protection in case of weak passwords, which happen to be the ones that are the most commonly used.

It would seem only logical that a service that is not using cryptographic primitives like hash functions correctly is more likely to get hacked and have its users' passwords or password hashes leaked. Those are often exposed publicly with no restrictions on access, and the internet turns out to be serving as a storage/medium.

That incidentally also means that anyone interested in their own compromise monitoring has at least *some* chances of successfully learning about their compromise, potentially a long time before it can be used to cause greater harm.

1.2 TLS

The Transport Layer Security protocol (or TLS) serves as as an encryption and *authentication* protocol to secure internet communications. An important part of the protocol is the *handshake*, during which the two communicating parties exchange messages that acknowledge each other's presence, verify each other, choose what cryptographic algorithms will be used and decide session keys. As there are multiple versions of the protocol in active duty even at the moment, the server together with the client need to agree upon the version they are going to use (it is recommended to use either v1.2 or v1.3 these days), pick cipher suites (if applicable), the client verifies the server's public key (and the signature of the certificate authority that issued it) and they both generate session keys for use after handshake completion.

TLSv1.3 dramatically reduced the number of available suites to only include the ones deemed secure enough, which is why it is no longer needed to manually specify what cipher suite should be used (or rely on the client/server to choose wisely). While possibly facing compatibility issues with legacy devices, the simplicity brought by enabling TLSv1.3 might be considered a worthy trade-off [15].

2 PASSWORDS

Passwords have been in use since the ancient times, apparently already the Roman sentries used passwords or *watchwords* to discern who was allowed to enter an area. The Roman army had a special system of distributing passwords among the encampment members on a wooden tablet. Fast forward a couple of thousand years, during the days of the Prohibition Era in the United States, it was the secret “speakeasies” that were protecting their illegitimate alcohol-serving business using passwords [16], [17]. During the World War II. the US paratroopers’ use of passwords has evolved to even include a counter-password.

According to McMillan, the first *computer* passwords date back to mid-1960s Massachusetts Institute of Technology (MIT), when researchers at the university built a massive time-sharing computer called CTSS. Apparently, *even then* the passwords did not protect the users as well as they were expected to [18].

Traditionally, passwords were expected to be memorised, but the large number of password-protected *services* these days can make this impractical. To list a few common examples, access to a bank account, electronic mailbox, personal computer encrypted disk are all protected by some form of a password.

A password still often consists of a *string* of characters typed into a prompt but its function is still the same: as per NIST it enables the *verifier* to infer the *claimant’s* identity via a secret the claimant holds.

There are always some arbitrary requirements applied to what the password can be, only some turn out to smarter than others.

Despite the impression given by the word “password”, it does not need to be an actual word, while a non-word (in the dictionary sense) may indeed be harder to guess, which is a desirable property of passwords. A memorized secret consisting of a sequence of words or other text separated by spaces is sometimes called a passphrase. A passphrase is similar to a password in usage, but the former is generally longer for added security.

2.1 Program-imposed constraints

Some of the following examples might be a bit anecdotal and more of an exception than a rule; nevertheless, when presented by a large-enough program creator/service

provider, their decisions reach a sufficient amount of population, enough that the author will call them influential. They form how users think when creating password and affect what users expect from other services they happen to visit and use from that point on, as well.

2.1.1 Short arbitrary length

It has been observed that a requirement for a “strong” password generally represents that a password is:

- longer than 7 characters,
- shorter than 11 characters,
- begins with a letter and ends with a number OR
- begins with a number and ends with a letter.

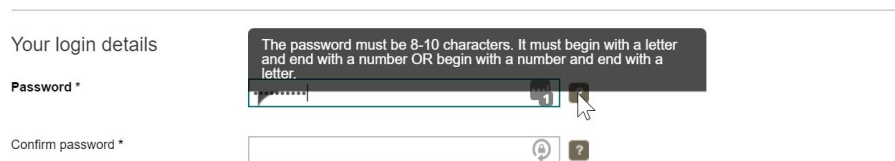


Figure 2.1 Short arbitrary password length limit

The error message in above the password input field depicted in Figure 2.1 is wrong for multiple reasons, and it is a classic example of short arbitrary length requirement [19]. It essentially prevents users from using passphrases, makes using a password manager impractical and all of that has apparently been done “because of security” [20]. Moreover, this might be an indicative of the fact that instead of storing passwords hashed (as it should be), they might be storing them in **plain text**. Otherwise, what reason could exist for the limit to be 10 characters? The recommendation of the US’s National Institute for Standards and Technology (NIST) in this regard is a minimum of 64 and a maximum of 256 characters, which, as they put it, *should be sufficient for most users’ needs*.

2.1.2 Restricting special characters

Service providers have too often been found forbidding the use of so called *special characters* in passwords for as long as passwords have been used to protect privileged

access. Ways of achieving the same may vary but the intent stays the same: preventing users from inputting characters into the system, which the system cannot comfortably handle, for “reasons”, which are usually something dubious along the lines of “an apostrophe may be used in SQL injection attacks” or “angle brackets may be used in XSS attacks”. Instead, the real message it often unwittingly announces is pointing right to the serious shortcomings of password handling of the site in question, as passwords should never be re-displayed in a context that is prone to Cross Site Scripting (XSS), and the passwords should always be hashed before being sent to the database anyway, leaving us with only alphanumeric characters, rendering the SQLi fears baseless.

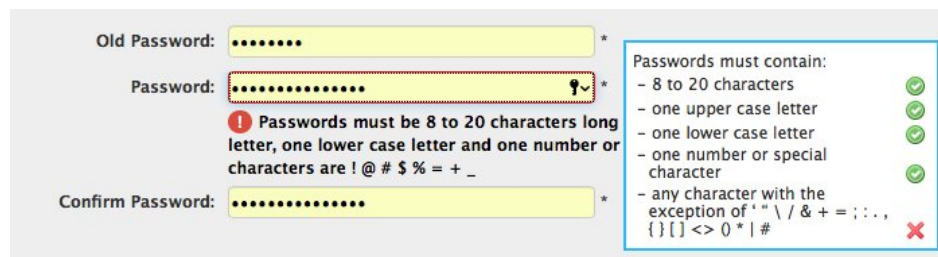


Figure 2.2 Forbidden special characters in passwords

Note that “Passw0rd!” would have been a perfectly acceptable password for the validator displayed in Figure 2.2 [21]. NIST’s recommendations on this matter are that all printing ASCII characters as well as the space character SHOULD be acceptable in memorized secrets, and Unicode characters SHOULD be accepted as well [22], [23].

2.1.3 Character composition requirements

There is a tendency to come up with bad passwords when there are character composition requirements in place, too. The reality is that instead of creating strong passwords directly, most users first try a basic version and then keep tweaking characters until the password ends up fulfilling the minimum requirement.

The *problem* is that that people use similar patterns, i.e. starting with capital letters, putting a symbol last and a number in the last two positions. This is also known to people cracking the password hashes and they run their dictionary attacks using the common substitutions, such as “\$” for “s”, “E” for “3”, “1” for “l”, “@” for “a” etc. [6], [7], [8]. It is safe to expect that the password created in this manner will almost certainly be bad, and the only achievement was to frustrate the user in order to still arrive at a bad password.

2.1.4 Other common issues

Some services don't allow users to paste into passwords fields (disabling them using JavaScript), thereby essentially breaking the password manager functionality, which is an issue because it encourages bad password practices such as weak passwords and likewise, password reuse.

Forced frequent password rotation is another common issue. Apparently, making frequent password rotations mandatory contributes to users developing a password creation *patterns*. Moreover, according to the British NCSC, the subject practice “carries no real benefits as stolen passwords are generally exploited immediately”, and the organisation calls it a modern-day security anti-pattern [24].

3 WEB SECURITY

The internet is a vast space full of intertwined concepts and ideas. It is a superset of the Web, even though the two terms often get conflated. However, not everything that is available on the internet can be accessed using web protocols and *resources*. This section delves into the concepts of web security.

3.1 Site Isolation

While website operators can perform steps to secure their sites, it is often the browsers holding the last line when these web servers are misconfigured, allowing the attacker to start exploiting a vulnerability in various ways.

Most users consume web content using web browsers. Modern browsers such as Firefox or Chromium are being built with a security focus in mind. Their developers are acutely aware of the dangers that parsing of untrusted code from the internet poses, which is precisely what the websites, the stylesheets and the accompanying scripts are.

This necessarily gets reflected in the way these programs are architected. Instead of the main, privileged browser process running everything directly, it spawns de-privileged child processes for each website. This extra line of defence should make it *harder for untrustworthy websites to access or steal information* from user accounts or other websites. Even if the misbehaving website does manage to “break some rules” within its own process, it should find it more difficult to steal data from other sites [25].

Firefox calls their version of *Site Isolation*-like functionality Project Fission, but the two are very similar, both in internal architecture and what they try to achieve [26]. Elements of the web page are scanned to decide whether they are allowed according to *same-site* restrictions and allocated shared or isolated memory based on the result.

Some Chromium users have been complaining on its high memory usage in the past, which might have been partially caused by Site Isolation user-protection features, unbeknownst to them.

3.2 Cross-site scripting

As per OWASP Top Ten list, injection is the third most observed issue across millions of websites. Cross-site scripting is a type of attack in which malicious code, such as

infected scripts, is injected into a website that would otherwise be trusted. Since the misconfiguration or a flaw of the application allowed this, the browser of the victim that trusts the website simply executes the code provided by the attacker. This code thus gains access to session tokens and any cookies associated with the website's origin, apart from being able to rewrite the HTML content. The results of XSS can range from account compromise to identity theft [27].

Solutions deployed against XSS vary. On the client side, it mainly comes down to good browser patching hygiene and, of course, avoiding sketchy websites is always a recommended practice. The security of the user is also to a degree reliant on browser features such as Site Isolation (see Section 3.1), and essentially browsers correctly parsing website directives such as the `X-Frame-Options`, `X-Content-Type-Options`, `X-Xss-Protection` and `Cross-Origin-Opener-Policy` HTTP headers. However, the latter falls flat if the website operators do not correctly configure their websites.

On the server side though, these options (indicating to the browsers *how* the site should be parsed) can directly be manipulated and configured. They should be fine-tuned to fit the needs of each specific website, as there is no one-size-fits-all in this case.

Furthermore, a new, powerful and comprehensive framework for controlling the admissibility of content has been devised more than 10 years ago now: Content Security Policy. Its capabilities superseded those of the previously mentioned options, and it is discussed more in-depth in the following section.

3.3 Content Security Policy

Content Security Policy (CSP) has been an important addition to the arsenal of website operators, even though not everybody has necessarily been utilising it properly or even taken notice. Once configured on the web server, it provides guarantees and employs protections against most common attack vectors on websites exactly where the websites are being parsed and displayed - in the (compliant) browser.

As per Weichselbaum et al. CSP is a mechanism designed to mitigate XSS [28], a long-lived king of the vulnerability lists [29]. It is a declarative policy mechanism that allows the website operator to decide what client-side resources can load on their website and what origins are among the permitted *sources* of content.

For example, dynamic content such as scripts can be restricted to only load from a list of trusted domains, and inline scripts can be blocked entirely, which is a huge win

against popular XSS techniques.

Not only that, scripts and stylesheets can also be allowed based on a cryptographic (SHA256, SHA384 or SHA512) hash of their content, which should be a known information to legitimate website operators prior to or at the time scripts are served, making sure no unauthorised script or stylesheet will ever be run on user's computer (running a compliant browser).

A policy of CSPv3, which is the current iteration of the concept, can be served either as a header or inside website's `<meta>` tag. Configuration is either site-wide or specific to each page.

Directive names are generally derived from the *sources* they are covering, and are thus often suffixed '-src', as in `script-src`, `img-src` or `style-src`, although some directives do not follow this pattern, `form-action`, `upgrade-insecure-requests` and `sandbox` representing this other group nicely.

Different directives are delimited using semicolon character at the end, and each directive can only appear once in the entire policy.

Special values exist for the origin website itself - `'self'` - and for disallowing *any* source - `'none'`.

A good policy is *targeted* and not overly broad. To give an example, a website that loads no JavaScript at all does not need to allow a popular CDN (Content Delivery Network) origin in its `script-src`, instead it should be set to `'none'`. CSP can also aid with clickjacking protection using its `frame-ancestors` directive, which can limit origins that have the permission to embed the website. This prevents the attacker from embedding the website at random places, for example malicious websites that masquerade as being legitimate, e.g. utilising 'Log in using service Xyz' frame, that in actuality just *pharms* the credentials.

Getting CSP right can be tricky depending on the nature of the site, but once grokked, it is relatively straight-forward and can increase the security of the site greatly. The recommended way to *test* CSP is to enable it in the *report-only* mode before turning it on in production.

CSP contains many more directives than could be mentioned in this section. Anybody interested is encouraged to have a read at <https://web.dev/csp/>.

4 CONFIGURATION

Every non-trivial program usually offers at least *some* way to tweak/manage its behaviour, and these changes are usually persisted *somewhere* on the filesystem of the host: in a local SQLite3 database, a *LocalStorage* key-value store in the browser, a binary or plain text configuration file. These configuration files need to be read and checked at least on program start-up and either stored into operating memory for the duration of the runtime of the program, or loaded and parsed, and the memory subsequently *freed* (initial configuration).

There is an abundance of configuration languages (or file formats used to craft configuration files, whether they were intended for it or not) available, TOML, INI, JSON, YAML, to name some of the popular ones (as of today).

Dhall stood out as a language that was designed with both security and the needs of dynamic configuration scenarios in mind, borrowing a concept or two from Nix (which in turn sources more than a few of its concepts from Haskell), and in its apparent core being very similar to JSON, which adds to a familiar feel [30], [31]. In fact, in Dhall's authors' own words it is: "a programmable configuration language that you can think of as: JSON + functions + types + imports" [32].

Among all the listed features, the especially intriguing one to the author was the promise of *types*. There are multiple examples directly on the project's documentation webpage demonstrating for instance the declaration and usage of custom types (that are, of course, merely combinations of the primitive types that the language provides, such as *Bool*, *Natural* or *List*, to name just a few), so it was not exceedingly hard to start designing a custom configuration *schema* for the program. Dhall, not being a Turing-complete language, also guarantees that evaluation *always* terminates eventually, which is a good attribute to possess for a configuration language.

4.1 Safety considerations

Having a programmable configuration language that understands functions and allows importing not only arbitrary text from random internet URLs, but also importing and *evaluating* (i.e. running) potentially untrusted code, it is important that there are some safety mechanisms employed, which can be relied on by the user. Dhall offers this in multiple features: enforcing a same-origin policy and (optionally) pinning a cryptographic hash of the value of the expression being imported.

4.2 Possible alternatives

While developing the program, the author has also come across certain shortcomings of Dhall, namely the long start-up on *cold cache*. It can generally be observed when running the program in an environment that does not allow persistently writing the cache files (a read-only filesystem), or does not keep the written cache files, such as a container that is not configured to mount persistent volumes to pertinent locations.

To describe the way Dhall works when performing an evaluation, it resolves every expression down to a combination of its most basic types (eliminating all abstraction and indirection) in the process called **normalisation** and then saves this result in the host's cache [33]. The `dhall-haskell` binary attempts to resolve the variable `${XDG_CACHE_HOME}` (have a look at *XDG Base Directory Spec* for details) to decide *where* the results of the normalisation will be written for repeated use [34]. Do note that this behaviour has been observed on a GNU/Linux host and the author has not verified this behaviour on another platforms, such as FreeBSD.

If normalisation is performed inside an ephemeral container (as opposed to, for instance, an interactive desktop session), the results effectively get lost on each container restart. That is both wasteful and not great for user experience, since the normalisation of just a handful of imports (which internally branches widely) can take an upwards of two minutes, during which the user is left waiting for the hanging application with no reporting on the progress or current status.

Workarounds for the above-mentioned problem can be devised relatively easily, but it would certainly *feel* better if there was no need to work *around* the configuration system of choice. For instance, bind mounting *persistent* volumes to pertinent locations inside the container (`${XDG_CACHE_HOME}/dhall,dhall-haskell`) would preserve cache between restarts. Alternatively, the cache could be pre-computed on container build (as the program is only expected to run with a compatible schema version, and that version *is* known at container build time for the supplied configuration).

Alternatives such as CUE (<https://cuelang.org/>) offer themselves nicely as an almost drop-in replacement for Dhall feature-wise, while also resolving the costly *cold cache* normalisation operations, which is in author's view Dhall's titular flaw. In a slightly contrasting approach, another emerging project called TySON (<https://github.com/jetpack-io/tyson>), which uses *a subset* of TypeScript to also create a programmable, strictly typed configuration language, opted to take a well-known language instead of reinventing the wheel, while still being able to retain feature parity with Dhall.

5 COMPROMISE MONITORING

There are, of course, several ways one could approach monitoring of compromised credentials, some more *manual* in nature than others. When using a service that is suspected/expected to be breached in the future, one can always create a unique username/password combination specifically for the subject service and never use that combination anywhere else. That way, if the credentials ever *do* happen to appear in a data dump online in the future, it is going to be a safe assumption as to where they came from.

Unfortunately, the task of actually *monitoring* the credentials can prove to be a little more arduous than one could expect at first. There are a couple of points that can prove to pose a challenge in case the search is performed by hand, namely:

- finding the breached data to look through
- verifying the trustworthiness of the data
- varying quality of the data
- sifting through (possibly) unstructured data by hand

Of course, as this is a popular topic for a number of people, the above-mentioned work has already been packaged into neat and practical online offerings. In case one decides in favour of using those, an additional range of issues (the previous one still applicable) arises:

- the need to trust the provider of the service with input credentials
- relying on the goodwill of the provider to be able to access the data
- hoping that the terms of service are kept as promised
- dependence on the quality and extent of their data sources

Besides that, there is a plethora of breaches floating around the Internet available simply as zip files, which makes the job of password compromise monitoring even harder.

The overarching goal of this thesis is devising and implementing a system in which the user can *monitor* whether their credentials have been *compromised* (at least as far as the data can tell), and allowing them to do so without needing to entrust their sensitive data to a provider.

5.1 Data Sources

A data source in this place is considered anything that provides the application with data that it understands.

Of course, the results of credential compromise verification/monitoring is only going to be as good as the data underpinning it, which is why it is imperative that high quality data sources be used, if at all possible. While great care does have to be taken to only choose the highest quality data sources, the application must offer a means to be able to utilise these.

The sources from which breached data can be loaded into an application can be split into two basic categories: **online** or **local**, and it is possible to further discern between them by whether the data they provide is *structured* or not.

An online source is generally a service that ideally exposes a programmatic API, which an application can query and from which it can request the necessary subsets of data. These types of services often additionally front the data by a user-friendly web interface for one-off searches, which is, however, not of use here.

Among some examples of online services could be named:

- Have I Been Pwned? - <https://haveibeenpwned.com>
- DeHashed - <https://dehashed.com>

Large lumps of unstructured data available on forums or shady web servers would technically also count here, given that they provide data and are available online. However, even though data is frequently found online precisely in this form, it is also not of direct use for the application without manual *preprocessing*, as it is attended to in Section 5.1.1.

Another source is then simply any locally supplied data, which, of course, could have been obtained from a breach available online beforehand.

Locally supplied data is specific in that it needs to be formatted in such a way that it is understood by the application. That is, the data supplied for importing cannot be in its original raw form anymore, instead it has to have been morphed into the precise shape the application needs for further processing. Once imported, the application can query the data at will, as it knows exactly the shape of it.

This supposes the existence of a *format* for importing, the schema of which is devised in Section 5.1.1.

5.1.1 Local Dataset Plugin

Unstructured breach data from locally available datasets can be imported into the application by first making sure it adheres to the specified schema (have a look at the breach `ImportSchema` in Listing 5.1). If it does not (which is very likely with random breach data, as already mentioned in Section 5.1), it needs to be converted to a form that *does* before importing it to the application, e.g. using a Python script or a similar method.

Attempting to import data that does not follow the outlined schema should result in an error. Equally so, importing a dataset which is over a reasonable size limit should by default be rejected by the program as a precaution. Unmarshaling, for instance, a 1 TiB document would most likely result in an out-of-memory (OOM) situation on the host running the application, assuming contemporary consumer hardware conditions (not HPC).

```
// ImportSchema is the model for importing locally available breach data.
type ImportSchema struct {
    Name           string
    Description    string
    Date           time.Time
    IsVerified     bool
    ContainsPasswords bool
    ContainsHashes bool
    HashType       string
    HashSalted     bool
    HashPeppered  bool
    ContainsUsernames bool
    ContainsEmails bool
    Data           *Data
}
```

Listing 5.1 Breach `ImportSchema` Go struct (imports from the standard library assumed)

The Go *struct* shown in Listing 5.1 will in actuality translate to a YAML document written and supplied by an administrative user of the program. And while the author

is personally not the greatest supporter of YAML; however, the format was still chosen for several reasons:

- relative ease of use (plain text, readability) for machines and people alike
- capability to store multiple *documents* inside of a single file
- most of the inputs being implicitly typed as strings
- support for inclusion of comments
- machine readability thanks to being a superset of JSON

The last point specifically should allow for documents similar to what can be seen in Listing 5.2 to be ingested by the program, read and written by humans and programs alike.

```
---
name: Horrible breach
date: 2022-04-23T00:00:00Z+02:00
description: impacted X in 2022, it contains 10 000 unique emails...
isVerified: false
containsPasswds: false
containsHashes: true
containsEmails: true
hashType: md5
hashSalted: false
hashPeppered: false
data:
  hashes:
    - hash1
    - hash2
    - hash3
  emails:
    - email1
    - ""
    - email3
---
# document #2, describing another breach.
name: Horrible breach 2
...
```

Listing 5.2 A YAML file containing breach data formatted according to the `ImportSchema`, optionally containing multiple documents

Notice how the emails list (`.data/emails`) in Listing 5.2 is missing one record, perhaps because it was mistakenly omitted due to either machine error or unfamiliarity with the format. This is a valid scenario (mistakes do happen) and the application needs to be account for it. Alternatively, the program could start dropping empty/partial records, but that behaviour could quickly lead to unhappy users. The golden rule for the program is to *always do the expected thing* (and also not being overly smart about it, i.e. the simpler program flow is often better).

5.1.2 Have I Been Pwned? Integration

Troy Hunt's **Have I Been Pwned?** online service (<https://haveibeenpwned.com/>) has been chosen as the online source of compromised data. The service offers public APIs, which were originally (and it was the intention of their author that they stay that way) provided free of charge and with little-to-no rate-limiting. A major overhaul in this regard has been revealed in November of 2022, where in addition to a new rate-limit system, different-levels-of-symbolic fees were introduced to obtain the API keys. These Apparently, the top consumers of the API seemed to utilise it orders of magnitude more than the average person, which led Hunt to devising a new, tiered API access system in which the *little guys* would not be subsidising the *big guys*. Additionally, the symbolic fee of \$3.50 a month for the entry-level 10 requests-per-minute API key was meant to serve as a small barrier for (mis)users with nefarious purposes, but pose practically no obstacle for *legitimate* users, which is entirely reasonable [35].

The application's `hibp` module and database representation (`schema.HIBPSchema`) attempts to model the values returned by this API and declare actions to be performed upon the data, which is what facilitates the breach search functionality in the program.

The architecture is relatively simple. Breach data, including title, date, description and tags are cached by the application on start-up, as this API is not authenticated. In order for the authenticated API to be called, the application administrator first needs to configure an API key for the HIBP service via the management interface. The user can then enter the desired query parameters and the application then constructs the API call that is sent to the authenticated API, and awaits the response. As the API is rate-limited (individually, based on the API key supplied), sending requests directly after receiving them from the users would likely pose an issue at high utilisation times, and would result in the application ending up unnecessarily throttled. Request sending thus needs to be handled in the backend by a requests scheduler, as well as appropriately in the UI.

After a response from the API server arrives, the application attempts to *bind* the returned data to the pre-programmed *model* for validation, before finally parsing it. If the data can be successfully validated, it is saved into the database as a cache and the search query is performed on the saved data. The result is then displayed to the user for browsing.

6 DEPLOYMENT RECOMMENDATIONS

It is, of course, recommended that the application runs in a secure environment although definitions of that almost certainly differ depending on who you ask. General recommendations would be either to effectively reserve a machine for a single use case - running this program - so as to dramatically decrease the potential attack surface of the host, or run the program isolated in a container or a virtual machine. Furthermore, if the host does not need management access (it is a deployed-to-only machine that is configured out-of-band, such as with a *golden* image/container or declaratively with Nix), then an SSH *daemon* should not be running in it, since it is not needed. In an ideal scenario, the host machine would have as little software installed as possible besides what the application absolutely requires.

System-wide cryptographic policies should target the highest feasible security level, if at all available (as is the case by default on e.g. Fedora), covering SSH, DNSSEC and TLS protocols. Firewalls should be configured and SELinux (kernel-level mandatory access control and security policy mechanism) running in *enforcing* mode, if available.

6.1 Transport security

User connecting to the application should rightfully expect for their data to be protected *in transit* (i.e. on the way between their browser and the server), which is what *Transport Layer Security* family of protocols [15] was designed for, and which is the underpinning of HTTPS. TLS utilises the primitives of asymmetric cryptography to let the client authenticate the server (verify that it is who it claims it is) and negotiate a symmetric key for encryption in the process named the *TLS handshake* (see Section 1.2 for more details), the final purpose of which is establishing a secure communications connection. The operator should configure the program to either directly utilise TLS using configuration or have it listen behind a TLS-terminating *reverse proxy*.

6.2 Containerisation

Whether containerised or not, the application needs runtime access to secrets such as cookie encryption and authentication keys, or the database connection string (containing database host, port, user, password/encrypted password, authentication method and database name). It is a relatively common practice to deliver secrets to programs in configuration files; however, environment variables should be preferred. The program

could go one step further and only accept certain secrets as environment variables.

While it is not impossible to run a process scheduler (such as SystemD) inside a container, containers are well suited for single-program workloads. The fact that the application needs persistent storage also begs the question of *how to run the database in the container?*. Should data be stored inside the ephemeral container, it could end up being very short-lived (wiped on container restart), and barring container root volume snapshotting, it could turn backing up of data into a chore, which are likely not the desired features in this case. Moreover, it is the opinion of the author that multiprocess scheduling would inordinately complicate the container set-up. Instead of running a single program per container, which also provides good amounts of isolation if done properly, running multiple programs in one container would likely do the opposite.

As per the above, a more *sane* thing to do is to store data externally using a proper persistent storage method, such as a database. With Postgres being the safe bet among database engines, the program should be able to handle Postgres' most common authentication methods, namely *peer*, *scram-sha-256* and raw *password*, although the *password* option should not be used in production, *unless* the database connection is protected by TLS [36]. In any case, using the *scram-sha-256* method is preferable [37]. One way to verify during development that authentication works as intended is the *Password generator for PostgreSQL* tool, which generates an encrypted string from a raw user input [38].

If the application wants to use the *peer* authentication method, it is up to the operator to supply the Postgres socket to the container (e.g. as a volume bind mount). Equally, the operator needs to make sure that the database is either running in a network that is also directly attached to the container or that there is a mechanism in place that routes the requests for the database hostname to the destination, unless a static IP configuration is used, which is also possible.

Practically every container runtime satisfies this use case with a container *name-based routing* mechanism, which inside *Pods* (in case of Podman/Kubernetes) or common default networks (that are both NAT-ted *and* routed) enables resolution of container names. This abstraction is a responsibility of specially configured (most often autoconfigured) pieces of software, Aardvark in case of Podman, and CoreDNS for Kubernetes, and it makes using short-lived containers in dynamic networks convenient.

7 SUMMARY

Passwords (and/or passphrases) are in use everywhere and will quite probably continue to be for the foreseeable future. If not as *the* principal way to authenticate, then at least as *a* way to authenticate. And for as long as passwords are going to be handled and stored, they *are* going to get leaked, be it due to user or provider carelessness, or the attackers' resolve and wit. Of course, sifting through the heaps of available password breach data by hand is not a reasonable option, and therefore tools providing assistance come in handy. The following part of this thesis will explore that issue and suggest a solution.

II. PRACTICAL PART

8 INTRODUCTION

A part of the task of this thesis was to build an actual application, which was named Password Compromise Monitoring Tool, or `pcmt` for short. Therefore, the development process, the general tools and practices as well as the specific outcome are all described in the following sections. A whole section is dedicated to application architecture, whereby relevant engineering choices are justified and motifs preceding the decisions are explained. This part then flows into recommendations for more of a production deployment and concludes by describing the validation methods chosen and used to ensure correctness and stability of the program.

8.1 Kudos

The program that has been developed as part of this thesis used and utilised a great deal of free (as in *freedom*) and open-source software in the process, either directly or as an outstanding work tool, and the author would like to take this opportunity to recognise that fact¹⁾.

In particular, the author acknowledges that this work would not be the same without:

- vim (<https://www.vim.org/>)
- Arch Linux (<https://archlinux.org/>)
- ZSH (<https://www.zsh.org/>)
- kitty (<https://sw.kovidgoyal.net/kitty/>)
- Nix (<https://nixos.org/explore.html>)
- pre-commit (<https://pre-commit.com/>)
- Podman (<https://podman.io/>)
- Go (<https://go.dev/>)

All the code was typed into VIM, the shell used was ZSH, and the terminal emulator of choice was kitty. The development machines ran a recent installation of *Arch Linux*²⁾ and Fedora 38, both using a 6.{2,3,4}.x XanMod variant of the Linux kernel.

¹⁾**Disclaimer:** the author is not affiliated with any of the projects mentioned on this page.

²⁾(by the way) <https://i.redd.it/mfrfqy66ey311.jpg>.

9 DEVELOPMENT

The source code of the project was being versioned since the start, using the popular and industry-standard git (<https://git-scm.com>) source code management (SCM) tool. Commits were made frequently and, if at all possible, consist of small and self-contained changes of code, trying to follow sane commit message *hygiene*, i.e. striving for meaningful and well-formatted commit messages. The name of the default branch is `development`, since that is what the author likes to choose for new projects that are not yet stable (it is in fact the default in author's `.gitconfig`).

9.1 Commit signing

Since git allows cryptographically *singing* all commits, it would be unwise not to take advantage of this. For the longest time, GPG was the only method available for signing commits in git; however, that is no longer applicable [39]. These days, it is also possible to both sign and verify one's git commits (and tags!) using SSH keys, namely those produced by OpenSSH, which *can* be the same ones that can be used to log in to remote systems. The author has, of course, not reused the same key pairs that are used to connect to machines for signing commits. A different, Ed25519 elliptic curve key pairs have been used specifically for signing. Public components of these keys are enclosed in this thesis as Appendix 1.1 for future reference.

The validity of a signature on a particular commit can be viewed with git using the following commands (the `%` sign denotes the shell prompt):

```
% cd <cloned project dir>
% git show --show-signature <commit>
% # alternatively:
% git verify-commit <commit>
```

Listing 9.1 Verifying the signature of a git commit

There is one caveat to this though, git first needs some additional configuration for the code in Listing 9.1 to work as one would expect. Namely that the public key used to verify the signature needs to be stored in git's "allowed signers file", then git needs to be told where that file is located using the configuration value `gpg.ssh.allowedsignersfile` and finally the configuration value of the `gpg.format` field needs to be set to `ssh`. Luckily, because git also allows the configuration values to be local to each repository, both of the mentioned issues can be solved by running the following commands from inside the cloned repository:

```
% # set the signature format for the local repository.
% git config --local gpg.format ssh
% # save the public key.
% cat > ./tmp-allowed_signers \
  <<<'surtur <insert literal surtur pubkey>
  leo <insert literal leo pubkey>'
% # set the allowed signers file path for the local repository.
% git config --local gpg.ssh.allowedsignersfile=./tmp-allowed_signers
```

Listing 9.2 Prepare allowed signers file and signature format for git

After the code in Listing 9.2 is run, everything from the Listing 9.1 should remain applicable for the lifetime of the repository or until git changes implementation of signature verification. The git `user.name` that can be seen on the commits in the **Author** field is named after the machine that was used to develop the program, since the author uses different signing keys on each machine. That way the committer machine can be determined post-hoc.

For future reference, git has been used in the version `git version 2.4{0,1,2}.x`.

9.2 Continuous Integration

To increase both the author’s and public confidence in the atomic changes made over time, it was attempted to thoroughly *integrate* them using a continuous integration (CI) service that was plugged into the main source code repository since the early stages of development. This, of course, was again self-hosted, including the workers. The tool of choice there was Drone (<https://drone.io>) and the “docker” runner (in fact it runs any OCI container) was used to run the builds.

The way this runner works is that it creates an ephemeral container for every pipeline step and executes given *commands* inside of it. At the end of each step, the container is discarded while the repository clone, which is mounted into each container’s `/drone/src`, is persisted between steps, allowing it to be cloned from *origin* only at the start of the pipeline and then shared for all the following steps, saving bandwidth, time and disk writes.

The entire configuration used to run the pipelines can be found in a file named `.drone.yml` at the root of the main source code repository. The workflow consists of four pipelines, which are run in parallel. Two main pipelines are defined to build the frontend assets, the `pcmt` binary and run tests on `x86_64` GNU/Linux targets, one for each of Alpine (version 3.1{7,8}) and Arch. These two pipelines are identical apart from OS-specific bits such as installing a certain package, etc. For the record, other

OS-architecture combinations were not tested.

A third pipeline contains instructions to build a popular static analysis tool called `golangci-lint`, which is a sort of meta-linter, bundling a staggering number of linters (linter is a tool that performs static code analysis and can raise awareness of programming errors, flag potentially buggy code constructs, or *mere* stylistic errors), from sources and then perform the analysis of project's codebase using the freshly built binary. If the result of this step is successful, a handful of code analysis services get pinged in the next steps to take notice of the changes to project's source code and update their metrics. Details can be found in the main Drone configuration file `.drone.yml` and the configuration for the `golangci-lint` tool itself (such as what linters are enabled/disabled and their configurations) can be found in the root of the repository in the file named `.golangci.yml`.

The fourth pipeline focuses on linting the `Containerfile` and building the container and pushing in to a public container registry, although the latter action is only performed on feature branches, *pull request* or *tag* events.

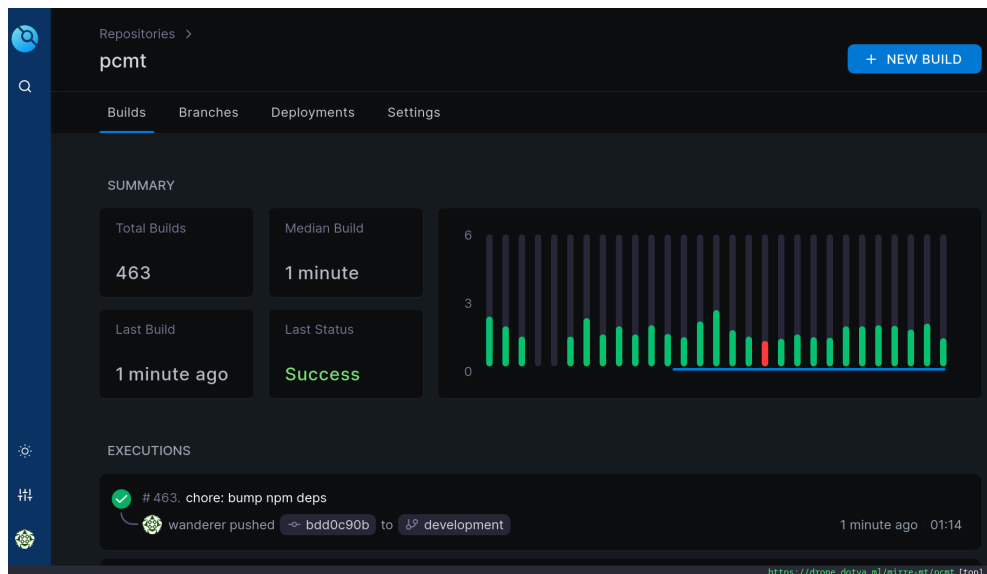


Figure 9.1 Drone CI median build time

The median build time as of writing was 1 minute, which includes running all four pipelines, and that is acceptable. Build times might of course vary depending on the hardware, for reference, these builds were run on a machine equipped with a Zen 3 Ryzen 5 5600 CPU with nominal clock times, DDR4 @ 3200 MHz RAM, a couple of PCIe Gen 4 NVMe drives in a mirrored setup (using ZFS) and a 600 Mbps downlink, software-wise running Arch with an author-flavoured Xanmod kernel version 6.{2,3,4}.x.

9.3 Source code repositories

The git repository containing source code of the `pcmt` project:

<https://git.dotya.ml/mirre-mt/pcmt.git>.

The git repository hosting the `pcmt` configuration schema:

<https://git.dotya.ml/mirre-mt/pcmt-config-schema.git>.

The repository containing the \LaTeX source code of this thesis:

<https://git.dotya.ml/mirre-mt/masters-thesis.git>.

All the pertaining source code was published in repositories on a publicly available git server operated by the author, the reasoning *pro* self-hosting being that it is the preferred way of guaranteed autonomy over one's source code, as opposed to large silos owned by big corporations having a track record of arguably not always deciding with user's best interest in mind (although recourse has been observed [40]). When these providers act on impulse or under public pressure they can potentially (at least temporarily) disrupt operations of their users. Thus, they are not only beholding their users to lengthy *terms of service* that *are subject to change at any given moment*, but also outside factors beyond their control. Granted, decentralisation can take a toll on discoverability of the project, but that is only a concern if rapid market penetration is a goal, not when aiming for an organically grown community.

9.4 Toolchain

Throughout the creation of this work, the *then-current* version of the Go programming language was used, i.e. `go1.20`.

To read more on why Go was chosen in particular, see Appendix 2.1. Equally, Nix and Nix-based tools such as `devenv` have also aided heavily during development, more on those is written in Appendix 2.2.

Table 9.2 contains the names and versions of the most important libraries and supporting software that were used to build the application.

Additionally, the dependency-version mapping for the Go program can be inferred from looking at the `go.mod`'s first *require* block at any point in time. The same can be achieved for *frontend* by glancing at the `package-lock.json` file.

Table 9.1 Tool/Library-Usage Matrix

Tool/Library	Usage
Go programming language	program core
Dhall configuration language	program configuration
Echo	HTTP handlers, controllers
ent	ORM using graph-based modelling
pq	Pure-Go Postgres drivers
bluemonday	sanitising HTML
TailwindCSS	utility-first approach to Cascading Style Sheets
PostgreSQL	persistent data storage

Table 9.2 Dependency-Version Matrix

Name	version
echo (https://echo.labstack.com/)	4.11.1
go-dhall (https://github.com/philandstuff/dhall-golang)	6.0.2
ent (https://entgo.io/)	0.12.3
pq (https://github.com/lib/pq/)	1.10.9
bluemonday (https://github.com/microcosm-cc/bluemonday)	1.0.25
tailwindcss (https://tailwindcss.com/)	3.3.0
PostgreSQL (https://www.postgresql.org/)	15.3

10 APPLICATION ARCHITECTURE

The application is written in Go and uses *gomodules*. The full name of the module is `git.dotya.ml/mirre-mt/pcmt`.

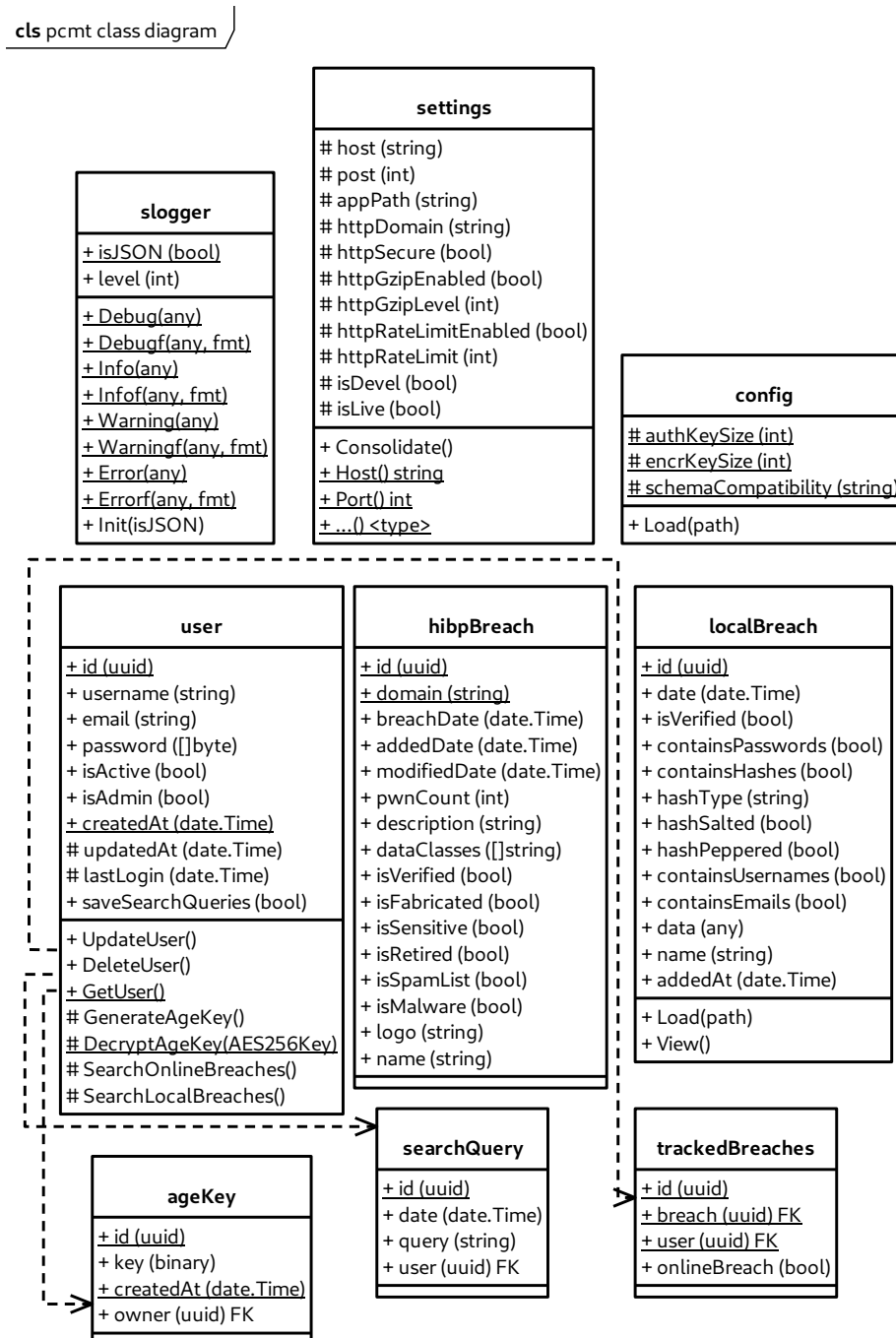


Figure 10.1 Application class diagram

10.1 Package structure

The source code of the module is organised into smaller, self-contained Go *packages* appropriately along a couple of domains: logging, core application, web routers, configuration and settings, etc. In Go, packages are delimited by folder structure – each folder can be a package.

Generally speaking, the program aggregates decision points into central places, such as `run.go`, which then imports child packages that facilitate each of the tasks of loading the configuration, connecting to the database and running migrations, consolidating flag, environment variable and configuration-based values into canonical *settings struct*, setting up web routes, authenticating requests, or handling `signals` and performing graceful shutdowns.

10.1.1 Internal package

The `internal` package was not used as of writing, but the author plans to eventually migrate *internal* logic of the program into the internal package to prevent accidental imports.

10.2 Logging

The program uses *dependency injection* to share a single logger instance (the same technique is also used to share the database client). This logger is then passed around as a pointer, so that the underlying data stays the same or is modified concurrently for all consumers. As a rule of thumb throughout the application, every larger `struct` that needs to be passed around is passed around as a pointer.

An experimental (note: not anymore, with `go1.21` it was brought into Go's *stdlib*) library for *structured* logging `slog` was used to facilitate every logging need that the program might have. It supports both JSON and plain-text logging, which was made configurable by the program. Either a configuration file value or an environment variable can be used to set this.

There are four log levels available by default (`DEBUG`, `INFO`, `WARNING`, `ERROR`) and the pertinent library functions are parametric. The first parameter of type `string` is the main message, that is supplied as a *value* to the *key* named appropriately `'msg'`, a feature

of structured loggers which can later be used for filtering. Any other parameters need to be supplied in pairs, serving as key and value, respectively.

This main `slog` interface has been extended in package `slogging` to also provide the formatting functionality of the `fmt` standard library package. This was achieved by directly embedding `slog.Logger` in a custom `struct` type named `Slogger` and implementing the additional methods on the custom type. The new type that embeds the original `slog.Logger` gets to keep its methods thanks to the composition nature of Go. Thus, common formatting directives like the one seen in Listing 10.1 are now supported with the custom logger, in addition to anything the base `slog.Logger` offers.

```
slogger.Debug("operation %q for user %q completed at %s", op, usr.ID, time.Now())
```

Listing 10.1 Example formatting expression supplied to the logger

Furthermore, functionality was added to support changing the log level at runtime, which is a convenient feature in certain situations.

10.3 Authentication

The authentication logic is relatively simple and its core has mostly been isolated into a custom *middleware*. User passwords are hashed using a secure KDF before ever being sent to the database. The KDF of choice is `bcrypt` (with a sane *Cost* of 10), which automatically includes *salt* for the password and provides “length-constant” time hash comparisons. The author plans to add support for the more modern `scrypt` and the state-of-the-art, P-H-C (Password Hashing Competition) winner algorithm `Argon2` (<https://github.com/P-H-C/phc-winner-argon2>) for flexibility.

10.4 SQLi prevention

No raw SQL queries are directly used to access the database, thus decreasing the likelihood of SQL injection attacks. Instead, parametric queries are constructed in code using a graph-like API of the `ent` library, which is attended to in depth in Section 11.3.

10.5 Configurability

Virtually any important value in the program has been made into a configuration value, so that the operator can customise the experience as needed. A choice of sane

configuration defaults was attempted, which resulted in the configuration file essentially only needing to contain secrets, unless there is a need to override the defaults. It is not entirely a *zero-config* situation, rather a *minimal-config* one. An example can be seen in Section 11.1.

Certain options deemed important enough (this was largely subjective) were additionally made into command-line *flags*, using the standard library package `flags`. Users wishing to display all available options can append the program with the `-help` flag, a courtesy of the mentioned `flags` package.

-host <hostname/IP> (string) Takes one argument and specifies the hostname, or the address to listen on.

-port <port number> (int) This flag takes one integer argument and specifies the port to listen on. The argument is validated at program start-up and the program has a fallback built in for the case that the supplied value is bogus, such as a string or a number outside the allowed TCP range 1 – 65535.

-printMigration A boolean option that, if set, makes the program print any **upcoming** database migrations (based on the current state of the database) and exit. The connection string environment variable still needs to be set in order to be able connect to the database and perform the schema *diff*. This option is mainly useful during debugging.

-devel This flag instructs the program to enter *devel mode*, in which all templates are re-parsed and re-executed upon each request, and the default log verbosity is changed to level `DEBUG`. Should not be used in production.

-import <path/to/file> (string) This option tells the program to perform an import of local breach data into program's main database. Obviously, the database connection string environment variable also needs to be present for this. The option takes one argument that is the path to file formatted according to the `ImportSchema` (consult Listing 5.1). The program prints the result of the import operation, indicating success or failure, and exits.

-version As could probably be inferred from its name, this flag makes the program to print its own version (that has been embedded into the binary at build time) and exit. A release binary would print something akin to a *semantic versioning*-compliant git tag string, while a development binary might simply print the truncated commit ID (consult `Containerfile` and `justfile`) of the sources used to build it.

10.6 Embedded assets

An important thing to mention is embedded assets and templates. Go has multiple mechanisms to natively embed arbitrary files directly into the binary during the regular build process. `embed.FS` from the standard library `embed` package was used to bundle all template files and web assets, such as images, logos and stylesheets at the module level. These are then passed around the program as needed, such as to the `handlers` package.

There is also a toggle in the application configuration (`LiveMode`), which instructs the program at start-up to either rely entirely on embedded assets, or pull live template and asset files from the filesystem. The former option makes the application more portable as it is wholly self-contained, while the latter allows for flexibility and customisation not only during development. Where the program looks for assets and templates in *live mode* is determined by another configuration options: `assetsPath` and `templatePath`.

10.7 Composability

The core templating functionality was provided by the `html/template` Go standard library package. Echo's `Renderer` interface has been implemented, so that template rendering could be performed directly using Echo's built-in facilities in a more ergonomic manner using `return c.Render(http.StatusOK, "home.tpl")`.

```
{{ if and .User .User.IsLoggedIn .User.IsAdmin }}  
...  
{{ end }}
```

Listing 10.2 Conditionally enabling functionality inside a Go template based on user access level

Templates used for rendering of the web pages were created in a composable manner, split into smaller, reusable parts, such as `footer.tpl` and `head.tpl`. Those could then be included e.g. using `{{ template "footer.tpl" }}`. Specific functionality is conditionally executed based on the determined level of access of the user, see Listing 10.2 for reference.

A popular HTML sanitiser `bluemonday` has been employed to aid with battling XSS. The program first runs every template through the sanitiser before rendering it, so that any user-controlled inputs are handled safely.

A dynamic web application should include a CSP configuration. The program therefore has the ability to calculate the hashes (SHA256/SHA384) of its assets (scripts, images) on the fly and it is able to use them inside the templates. This unlocks potentially using third party assets without opening up CSP with directives like `script-src 'unsafe-hashes'`. It also means that there is no need to maintain a set of customised head templates with pre-computed hashes next to script sources, since the application can perform the necessary calculations in user's stead.

10.8 Server-side rendering

The application constructs the web pages *entirely* on the server side, and it runs without a single line of JavaScript, of which the author is especially proud. It improves load times, decreases the attack surface, increases maintainability and reduces cognitive load that is required when dealing with JavaScript. Of course, that requires extensive usage of non-semantic `POST` requests in web forms even for data *updates* (where HTTP `PUTs` should be used) and the accompanying frequent full-page refreshes, but that still is not enough to warrant the use of JavaScript.

10.9 Frontend

Frontend-wise, the application Tailwind was used for CSS. It promotes the usage of flexible *utility-first* classes in the HTML markup instead of separating out styles from content. Understandably, this is somewhat of a preference issue and the author does not hold hard opinions in either direction; however, it has to be noted that this approach empirically allows for rather quick UI prototyping. Tailwind was chosen for having a reasonably detailed documentation and offering built-in support for dark/light mode, and partially also because it *looks* nice.

The Go templates containing the CSS classes need to be parsed by Tailwind in order to produce the final stylesheet that can be bundled with the application. The upstream provides an original CLI tool (`tailwindcss`), which can be used exactly for that action. Simple and accessible layouts were overall preferred, a single page was rather split into multiple when becoming convoluted. Data-backed efforts were made to create reasonably contrasting pages.

10.9.1 Frontend experiments

As an aside, the author has briefly experimented with WebAssembly to provide client-side dynamic functionality for this project, but has ultimately scrapped it in favour of the entirely server-side rendered approach. It is possible that it would get revisited in the future if necessary. Even from the short experiments it was obvious how much faster WebAssembly was when compared to JavaScript.

10.10 User isolation

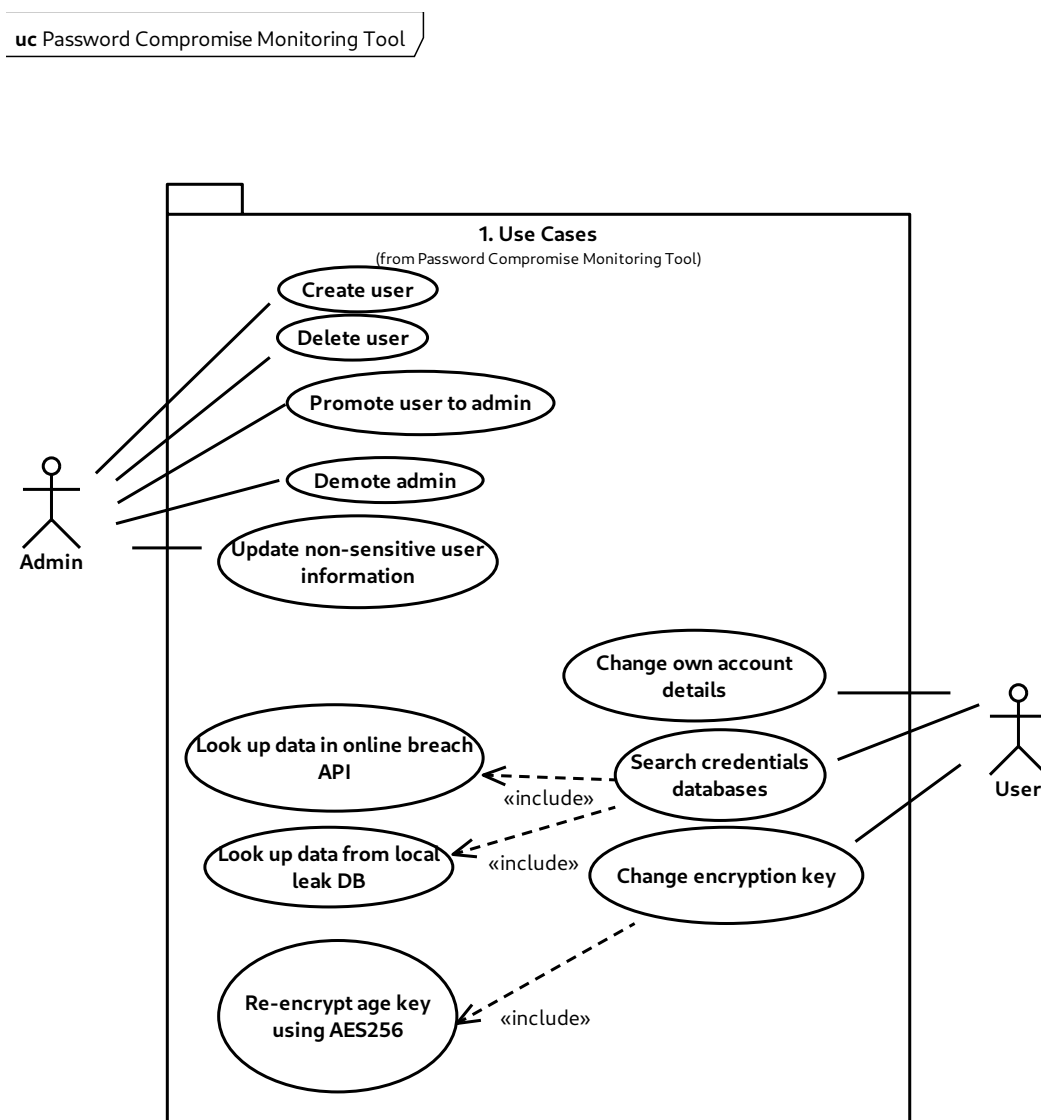


Figure 10.2 Application use case diagram

Users are only allowed into specific parts of the application based on the role they currently possess (Role-based Access Control).

While this short list might get amended in the future, initially only two basic roles were envisioned:

- Administrator
- User

It is paramount that the program protects itself from the insider threats as well, and therefore each role is only able to perform actions that it is explicitly assigned. While there definitely is a certain overlap between the capabilities of the two outlined roles, each also possesses unique features that the other one does not.

For instance, the administrator role is not able to perform breach data searches directly, for that a separate *user* account has to be devised. Similarly, a regular user is not able to manage breach lists and other users, because that is a privileged operation.

In-application administrators are not able to view (any) sensitive user data and should therefore only be able to perform the following actions:

- Create user accounts
- View user listing
- View user details
- Change user details, including administrative status
- Delete user accounts
- Refresh breach data from online sources

Let us consider a case when a user performs an operation on their own account. While demoting from administrator to a regular user should be permitted, promoting self to be an administrator would constitute a *privilege escalation* and likely be a precursor to at least a *denial of service* of sorts, as there would be nothing preventing the newly-admined user from disabling the accounts of all other administrators.

10.11 Zero trust principle and confidentiality

The program only sets generic titles (Settings, Home, Search) and thus foregoes disclosing information that would make it to browsers history.

There is no way for the application (and consequently, the in-application administrator) to read user's data (such as saved search queries). This is possible by virtue of encrypting the pertinent data before saving them in the database by a state-of-the-art `age` tool (backed by X25519) [41], [42]. The `age identity` itself is in turn encrypted by a passphrase that only the user controls. Of course, the user-supplied password is run by a password based key derivation function (`argon2`, version `id` with the officially recommended configuration parameters) before letting it encrypt *anything*.

The `age identity` is only generated once the user changes their password for the first time, in an attempt to prevent scenarios like the in-application administrator with access to physical database being able to both **recover** the key from the database and **decrypt** it, given that they already know the user password (because they set it when they created the user), which would subsequently give them unbounded access to any future encrypted data, as long as they would be able to maintain their database access. This is why generating the `age identity` is bound to the first password change.

Of course, the supposed evil administrator could simply perform the password change themselves! However, the user would at least be able to find those changes in the activity logs and know to *not* use the application under such circumstances. But given the scenario of a total database compromise, the author finds that all hope is *already* lost at that point (similar to physical access to a computer). At least when the database is dumped, it should only contain non-sensitive, functional information in plain text, everything else should be encrypted.

Consequently, both the application operators and the in-application administrators should never be able to learn the details of what the user is searching for, the same being by extension partly applicable even to potential attackers with direct access to the database. Thus, the author maintains that a scenario, which could potentially lead to a breach (apart from a compromised actual password) would have to entail some form of operating memory acquisition on the machine hosting the application, for instance using `LiME` [43], or perhaps directly the *hypervisor*, if considering a virtualised (“cloud”) environments. Alternatively, all but one (memory acquisition) of the above issues could perhaps be remedied by simply not storing any user queries, turning off informative logging, and only letting the program be mediate the data sources.

11 IMPLEMENTATION

11.1 Dhall Configuration Schema

The configuration schema was at first being developed as part of the main project's repository, before it was determined that both the development and overall clarity would benefit from the schema living in its own repository (see Section 9.3 for details). This enabled the schema to be independently developed and versioned, and only be pulled into the main application whenever it was determined to be ready.

```
let Schema =
  { Type =
    { Host : Text
    , Port : Natural
    , HTTP :
      { Domain : Text
      , Secure : Bool
      , AutoTLS : Bool
      , TLSKeyPath : Text
      , TLSCertKeyPath : Text
      , HSTSMAXAGE : Natural
      , ContentSecurityPolicy : Text
      , RateLimit : Natural
      , Gzip : Natural
      , Timeout : Natural
      }
    , Mailer :
      { Enabled : Bool
      , Protocol : Text
      , SMTPAddr : Text
      , SMTPPort : Natural
      , ForceTrustServerCert : Bool
      , EnableHELO : Bool
      , HELOHostname : Text
      , Auth : Text
      , From : Text
      , User : Text
      , Password : Text
      , SubjectPrefix : Text
      , SendPlainText : Bool
      }
    , LiveMode : Bool
    , DevelMode : Bool
    , AppPath : Text
    , Session :
      { CookieName : Text
      , CookieAuthSecret : Text
      , CookieEncrSecret : Text
      , MaxAge : Natural
      }
    , Logger : { JSON : Bool, Fmt : Optional Text }
    , Init : { CreateAdmin : Bool, AdminPassword : Text }
    , Registration : { Allowed : Bool }
    }
  }
```

Listing 11.1 Dhall configuration schema version 0.0.1-rc.2

Full schema with type annotations can be seen in Listing 11.1.

The `let` statement declares a variable called `Schema` and assigns to it the result of the expression on the right side of the equals sign, which has for practical reasons been trimmed and is displayed without the *default* block. The default block is instead shown in its own Listing 11.2.

The main configuration comprises both raw attributes and child records, which allow for grouping of related functionality. For instance, configuration settings pertaining mailserver setup are grouped in a record named **Mailer**. Its attribute **Enabled** is annotated as **Bool**, which was deemed appropriate for an on-off switch-like functionality, with the only permissible values being either *True* or *False*.

Do note that in Dhall `true != True`, since internally **True** is a `Bool` constant built directly into Dhall (see “The Prelude” for reference), while `true` is evaluated as an *unbound* variable, that is, a variable *not* defined in the current *scope* and thus not *present* in the current scope [44].

Another one of Dhall’s specialties is that ‘==’ and ‘!=’ (in)equality operators **only** work on values of type `Bool`, which for example means that variables of type `Natural` (`uint`) or `Text` (`string`) cannot be compared directly as is the case in other languages. That either leaves the comparing work for a higher-level language (such as Go). Alternatively, from the perspective of the Dhall authors *enums* are the promoted way to solve this when the value matters, i.e. derive a custom *named* type from a primitive type and compare *that*.

```

, default =
  -- | have sane defaults.
  { Host = ""
  , Port = 3000
  , HTTP =
    { Domain = ""
    , Secure = False
    , AutoTLS = False
    , TLSKeyPath = ""
    , TLSCertKeyPath = ""
    , HSTSMaxAge = 0
    , ContentSecurityPolicy = ""
    , RateLimit = 0
    , Gzip = 0
    , Timeout = 0
    }
  , Mailer =
    { Enabled = False
    , Protocol = "smtps"
    , SMTPAddr = ""
    , SMTPPort = 465
    , ForceTrustServerCert = False
    , EnableHELO = False
    , HELOHostname = ""
    , Auth = ""
    , From = ""
    , User = ""
    , Password = ""
    , SubjectPrefix = "pcmt - "
    , SendPlainText = True
    }
  , LiveMode =
    -- | LiveMode controls whether the application looks for
    -- | directories "assets" and "templates" on the filesystem or
    -- | in its bundled Embed.FS.
    False
  , DevMode = False
  , AppPath =
    -- | AppPath specifies where the program looks for "assets" and
    -- | "templates" in case LiveMode is True.
    "."
  , Session =
    { CookieName = "pcmt_session"
    , CookieAuthSecret = ""
    , CookieEncrSecret = ""
    , MaxAge = 3600
    }
  , Logger = { JSON = True, Fmt = None Text }
  , Init =
    { CreateAdmin =
      -- | if this is True, attempt to create a user with admin
      -- | privileges with the password specified below
      False
    , AdminPassword =
      -- | used for the first admin, forced change on first login.
      "50ce50fd0e4f5894d74c4caecb450b00c594681d9397de98ffc0c76af5cff5953eb795f7"
    }
  , Registration.Allowed = True
  }
}

```

in Schema

Listing 11.2 Dhall configuration defaults for schema version 0.0.1-rc.2

11.2 Data integrity and authenticity

The user can interact with the application via a web client, such as a browser, and is required to authenticate for all sensitive operations. To not only know *who* the user is but also make sure they are *permitted* to perform the action they are attempting, the program employs an *authorisation* mechanism in the form of sessions. These are on the client side represented by cryptographically signed and encrypted (using 256-bit AES) HTTP cookies. That lays foundations for a few things: the data saved into the cookies can be regarded as private because short of future *quantum computers* only the program itself can decrypt and access the data, and the data can be trusted since it is both signed using the key that only the program controls and *encrypted* with *another* key that equally only the program holds.

The cookie data is only ever written *or* read at the server side, solidifying the authors decision to let it be encrypted, as there is no point in not encrypting it for some perceived client-side simplification. Users navigating the website send their session cookie (if it exists) with **every request** to the server, which subsequently verifies the integrity of the data and in case it is valid, determines the existence and potential amount of user privilege that should be granted. Public endpoints do not mandate the presence of a valid session by definition, while at protected endpoints the user is authenticated at every request. When a session expires or if there is no session to begin with, the user is either shown a *Not found* error message, the *Unauthorised* error message or redirected to `/signin`, depending on the endpoint or resource, as can be seen, this behaviour is not uniform and depends on the resource and/or the endpoint.

Another aspect that contributes to data integrity from *another* point of view is utilising database *transactions* for bundling together multiple database operations that collectively change the *state*. Using the transactional jargon, the data is only *committed* if each individual change was successful. In case of any errors, the database is instructed to perform an atomic *rollback*, which brings it back to a state before the changes were ever attempted.

The author has additionally considered the thought of utilising an embedded immutable database like immudb (<https://immudb.io>) for record keeping (verifiably storing data change history) and additional data integrity checks, e.g. for tamper protection purposes and similar; however, that work remains yet to be materialised.

11.3 Database schema

The database schema is not being created by manually typing out SQL statements. Instead, an Object-relational Mapping (ORM) tool named `ent` is used, which allows defining the table schema and relations entirely in Go. The upside of this approach is that the *entity* types are natively understood by code editors, and they also get type-checked by the compiler for correctness, preventing all sorts of headaches and potential bugs.

Since `ent` encourages the usage of *declarative migrations* at early stages of the project, it is not required for the database schema to exist on application start-up in form of raw SQL (or HCL). Instead, `ent` only requires a valid connection string providing reasonably privileged access to the database and it handles the database configuration by auto-generating SQL with the help of the companion embedded library `Atlas` (<https://atlasgo.io/>). The upstream project (`ent`) encourages moving to otherwise more traditional *versioned migrations* for more mature projects, so that is on the roadmap for later.

The best part about using `ent` is that there is no need to define supplemental methods on the models, as with `ent` these are meant to be *code generated* (in the older sense of word, not with Large Language Models) into existence. Code generation creates files with actual Go models based on the types of the attributes in the database schema model, and the respective relations are transformed into methods on the receiver or functions taking object attributes as arguments.

For instance, if the model's attribute is a string value `Email`, `ent` can be used to generate code that contains methods on the user object like the following:

- `EmailIn(pattern string)`
- `EmailEQ(email string)`
- `EmailNEQ(email string)`
- `EmailHasSuffix(suffix string)`

These methods can further be imported into other packages and this makes working with the database a morning breeze.

All the database *entity* IDs were declared as type `UUID` (*universally unique ID, theoretically across space and time*), contrary to the more traditional *integer* IDs.

Support for UUIDs was provided natively by the supported databases and in Go via a popular and vetted open-source library (github.com/google/uuid). Among the upsides of using UUIDs over integer IDs is that there is no need to manually increment the ID. But more importantly, there is also the fact that compared to 32-bit¹⁾ signed integers the UUID is a somewhat randomly generated 16 byte (128 bit) array, reducing chances of collision.

Barring higher chances of preventing conflicts during imports of foreign databases, this design decision might not provide any advantage for the current system *at the moment*. It could, however, hold importance in the future, should the database ever be deployed in a replicated, high-availability (HA) manner with more than one concurrent *writer* (replicated application instances).

The relations between entities as modelled with `ent` can be imagined as the edges connecting the nodes of a directed *graph*, with the nodes representing the entities. This conceptualisation lends itself to a more human-friendly querying language, where the directionality can be expressed with words describing ownership, like so:

```
descr, err := users.Query().
    Where().
    LocalBreach.
    Has(BreachDetailXyz).
    Has(Description).
    Only(ctx)
```

Listing 11.3 Ent graph query

¹⁾In Go, integer size is architecture dependent, see https://go.dev/ref/spec#Numeric_types.

12 DEPLOYMENT

A deployment set-up, as suggested in Section 6, is already *partially* covered by the multi-stage `Containerfile` that is available in the main sources. Once built, the resulting container image only contains a handful of things it absolutely needs:

- a self-contained statically linked copy of the program
- a default configuration file and corresponding Dhall expressions cached at build time
- a recent CA certs bundle

Since the program also needs a database for proper functioning, an example scenario includes the application container being run in a Podman **pod** (as in a pea pod or pod of whales) together with the database. That results in not having to expose the database to the entire host or out of the pod at all, it is only available over pod's `localhost`. Hopefully it goes without saying that the default values of any configuration secrets should be substituted by the application operator with new, securely generated ones (read: using `openssl rand` or `pwgen`).

12.1 Rootless Podman

Assuming rootless Podman set up and the `just` tool installed on the host, the application could be deployed by following a series of relatively simple steps:

- build (or pull) the application container image
- create a pod with user namespacing, exposing the application port
- run the database container inside the pod
- run the application inside the pod

In concrete terms, it would resemble something along the lines of Listing 12.1. Do note that all the commands are executed under the unprivileged `user@containerHost` that is running rootless Podman, i.e. it has `UID/GID` mapping entries in `/etc/setuid` and `/etc/setgid` files **prior** to running any Podman commands.

```

# From inside the project folder, build the image locally using kaniko.
just kaniko

# Create a pod, limit the amount of memory/CPU available to its containers.
podman pod create --replace --name pcmt \
  --memory=100m --cpus=2 \
  --userns=keep-id -p3005:3000

# Create the database folder and run the database in the pod.
mkdir -pv ./tmp/db
podman run --pod pcmt --replace -d --name "pcmt-pg" --rm \
  -e POSTGRES_INITDB_ARGS="--auth-host=scram-sha-256 \
    --auth-local=scram-sha-256" \
  -e POSTGRES_PASSWORD=postgres \
  -v $PWD/tmp/db:/var/lib/postgresql/data:Z \
  --health-cmd "sh -c 'pg_isready -U postgres -d postgres'" \
  --health-on-failure kill \
  --health-retries 3 \
  --health-interval 10s \
  --health-timeout 1s \
  --health-start-period=5s \
  docker.io/library/postgres:15.2-alpine3.17

# Run the application itself in the pod.
podman run --pod pcmt --replace --name pcmt-og -d --rm \
  -e PCMT_LIVE=False \
  -e PCMT_DBTYPE="postgres" \
  -e PCMT_CONNSTRING="host=pcmt-pg port=5432 sslmode=disable \
    user=postgres dbname=postgres password=postgres" \
  -v $PWD/config.dhall:/config.dhall:Z,ro \
  docker.io/immawanderer/mt-pcmt:testbuild -config /config.dhall

```

Listing 12.1 Example application deployment using rootless Podman

To summarise Listing 12.1, first the application container is built from inside the project folder using `kaniko`. The container image could alternatively be pulled from the container repository, but it makes more sense showing the image being built from sources with the listing depicting a `:testbuild` tag being used.

Next, a *pod* is created and given a name, setting the port binding for the application. Then, the database container is started inside the pod, configured with a healthchecking mechanism.

As a final step, the application container itself is run inside the pod. The application configuration named `config.dhall` located in `$PWD` is mounted as a volume into container's `/config.dhall`, providing the application with a default configuration. The default container does contain a default configuration for reference, however, running the container without additionally providing the necessary secrets would fail.

12.1.1 Sanity checks

Also do note that the application connects to the database using its *container* name, i.e. not the IP address. This is possible thanks to Podman setting up DNS resolution inside pods using default networks in such a way that all containers in the pod can reach each other using their (container) names.

Interestingly, connecting via `localhost` from containers inside the pod would also work. Inside the pod, any container in the pod can reach any other container in the same pod via *pod's* own `localhost`, thanks to a shared network name space [45].

In fact, *pinging* (sending ICMP packets using the `ping` command) the database and application containers from an ad-hoc Alpine Linux container that just joined the pod temporarily yields:

```
user@containerHost % podman run --rm -it \  
  --user=0 \  
  --pod=pcmt \  
  docker.io/library/alpine:3.18  
/ % ping -c2 pcmt-og  
PING pcmt-og (127.0.0.1): 56 data bytes  
64 bytes from 127.0.0.1: seq=0 ttl=42 time=0.072 ms  
64 bytes from 127.0.0.1: seq=1 ttl=42 time=0.118 ms  
  
— pcmt-og ping statistics —  
2 packets transmitted, 2 packets received, 0% packet loss  
round-trip min/avg/max = 0.072/0.095/0.118 ms  
/ % ping -c2 pcmt-pg  
PING pcmt-pg (127.0.0.1): 56 data bytes  
64 bytes from 127.0.0.1: seq=0 ttl=42 time=0.045 ms  
64 bytes from 127.0.0.1: seq=1 ttl=42 time=0.077 ms  
  
— pcmt-pg ping statistics —  
2 packets transmitted, 2 packets received, 0% packet loss  
round-trip min/avg/max = 0.045/0.061/0.077 ms  
/ %
```

Listing 12.2 Pinging pod containers using their names

Was the application deployed in a traditional manner instead of using Podman, the use of FQDNs or IPs would be probably be necessary, as there would be no magic resolution of container names happening transparently in the background.

12.1.2 Database isolation from the host

A keen observer has undoubtedly noticed that the pod constructed in Listing 12.1 did only create the binding for a port used by the application (5005/tcp). The Postgres default port 5432/tcp is not among pod's port bindings, as can be seen in the pod creation command in the said listing. This can also easily be verified using the command in Listing 12.3:

```
user@containerHost % podman pod inspect pcmt \
  --format="Port bindings: {{.InfraConfig.PortBindings}}\n\
  Host network: {{.InfraConfig.HostNetwork}}"
Port bindings: map[3000/tcp:[{ 5005}]]
Host network: false
```

Listing 12.3 Podman pod port binding inspection

To be absolutely sure that the database is available only internally in the pod (unless, of course, there is another process listening on the subject port), and that connecting to the database from outside the pod (i.e. from the container host) really *does* fail, the following commands can be issued:

```
user@containerHost % curl localhost:5432
→ curl: (7) Failed to connect to localhost port 5432 after 0 ms: Couldn't
  connect to server
```

Listing 12.4 In-pod database is unreachable from the host

The error in Listing 12.4 is indeed expected, as it is the result of the database port not being exposed from the pod.

Of course, since a volume (essentially a bind mount) from the host is used, the actual data is still accessible on the host, both to privileged users and the user running the pod. On the host with SELinux support, the `:Z` volume addendum at least ensures that the content of the volume is directly inaccessible to other containers, including the application container running inside the same pod, via SELinux labelling.

12.1.3 Health checks

Running the containers with health checks can be counted among the few crucial settings. That way the container runtime can periodically *check* that the application running inside the container is behaving correctly and instructions can be provided on what action should be taken, should the health of the application evaluate unsatisfyingly. Furthermore, different sets of health checking commands can be passed with Podman for start-up and runtime.

12.2 Reverse proxy configuration

If the application is deployed behind a reverse proxy, such as NGINX, the configuration snippet in Listing 12.5 might apply. Do note how the named upstream server `pcmt` references the port that was exposed from the pod created in Listing 12.1.

```
upstream pcmt {
    server 127.0.0.1:5005;
}
server {
    return 301 https://<pcmt domain>$request_uri;
    listen 80;
    listen [::]:80;
    server_name <pcmt domain> www.<pcmt domain>;
    return 404;
    add_header Referrer-Policy "no-referrer, origin-when-cross-origin";
}
server {
    server_name <pcmt domain>;
    access_log /var/log/nginx/<pcmt domain>.access.log;
    error_log /var/log/nginx/<pcmt domain>.error.log;
    location / {
        proxy_pass http://pcmt;
        proxy_set_header X-Forwarded-Host $host;
        proxy_set_header X-Forwarded-For $proxy_add_forwarded_for;
    }
    location /robots.txt {
        allow all;
        add_header Content-Type "text/plain; charset=utf-8";
        add_header X-Robots-Tag "all, noarchive, notranslate";
        return 200 "User-agent: *\nDisallow: /";
    }
    include sec-headers.conf;

    add_header X-Real-IP $remote_addr;
    add_header X-Forwarded-For $proxy_add_x_forwarded_for;
    add_header X-Forwarded-Proto $scheme;
    more_set_headers 'Early-Data: $ssl_early_data';

    listen [::]:443 ssl http2;
    listen 443 ssl http2;
    ssl_certificate /etc/letsencrypt/live/<pcmt domain>/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/<pcmt domain>/privkey.pem;
    include /etc/letsencrypt/options-ssl-nginx.conf;
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem;

    # reduce TTFB
    ssl_buffer_size 4k;
}
```

Listing 12.5 Example reverse proxy configuration snippet

The snippet describes how traffic arriving at port `80/tcp` (IPv4 or IPv6) that matches the domain name(s) `{www.,}<pcmt domain>` (`<pcmt domain>` being the domain name that the program was configured with, including appropriate DNS records) gets 301-redirected to the same location (`$request_uri`), only over HTTPS. If the server name does not match, a 404 is returned instead. In the main location block, all traffic except for `/robots.txt` is forwarded to the named backend, with headers added on top by the proxy in order to label the incoming requests as *not* originating at the proxy. The *robots* route is treated specially, immediately returning a directive that disallows crawling of any resource on the page for all. The proxy is also instructed to log access and error events to specific log files, finally load the domain's TLS certificates (obtained out of band), reduce the `ssl_buffer_size` and listen on port `443/tcp` (dual stack).

13 VALIDATION

13.1 Unit tests

Unit testing is a hot topic for many people and the author does not count himself to be a staunch supporter of neither extreme. The “no unit tests” opinion seems to discount any benefit there is to unit testing, while a “TDD-only”¹⁾ approach can be a little too much for some people’s taste. The author tends to prefer a *middle ground* approach in this particular case, i.e. writing enough tests where meaningful, but not necessarily testing everything or writing tests prior to business logic code. Arguably, following the practice of TDD should result in writing a *better designed* code, particularly because there needs to be a prior thought about the shape and function of the code, as it is tested for before being even written, but it adds a slight inconvenience to what is otherwise a straightforward process.

Thanks to Go’s built in support for testing via its `testing` package and the tooling in the `go` tool, writing tests is relatively simple. Go looks for files in the form `<filename>_test.go` in the present working directory but can be instructed to look for test files in packages recursively found on any path using the ellipsis, like so: `go test ./path/to/package/...`, which then *runs* all the tests found, and reports some statistics, such as the time it took to run the test or whether it succeeded or failed. To be precise, the test files also need to contain test functions, which are functions with the signature `func TestWhatever(t *testing.T){}` and where the function prefix “Test” is just as important as the signature. Without it, the function is not considered to be a testing function despite having the required signature and is therefore *not* executed during testing.

This test lookup behaviour, however, also has a neat side effect: all the test files can be kept side-by-side their regular source counterparts, there is no need to segregate them into a specially blessed `tests` folder or similar, which in author’s opinion improves readability. As a failsafe, in case no actual test are found, the current behaviour of the tool is to print a note informing the developer that no tests were found, which is handy to learn if it was not intended/expected. When compiling regular source code, the Go files with `_test` in the name are simply ignored by the build tool.

¹⁾TDD, or Test Driven Development, is a development methodology whereby tests are written *first*, then a complementary piece of code that is supposed to be tested is added, just enough to get past the compile errors and to see the test *fail* and then is the code finally refactored to make the test *pass*. The code can then be fearlessly extended because the test is the safety net catching the programmer when the mind slips and alters the originally intended behaviour of the code.

13.2 Integration tests

Integrating with external software, namely the database in case of this program, is designed to utilise the same mechanism that was mentioned in the previous section: Go's `testing` package. These tests verify that the code changes can still perform the same actions with the external software that were possible before the change and are run before every commit locally and then after pushing to remote in the CI.

13.2.1 `func TestUserExists(t *testing.T)`

In the integration test shown in Listing 13.1, it is prefaced at line 10 by declaring a helper function `getCtx() context.Context`, which takes no arguments and returns a new `context.Context` initialised with the value of the global logger. As previously mentioned, that is how the logger gets injected into the user module functions. The actual test function with the signature `TestUserExists(t *testing.T)` defines a database connection string at line 21 and attempts to open a connection to the database. The database in use here is SQLite3 running in memory mode, meaning no file is actually written to disk during this process. Since the testing data is not needed after the test, this is desirable. Next, a defer statement calls the `Close()` method on the database object, which is the Go idiomatic way of closing files and network connections (which are also an abstraction over files on UNIX-like operating systems such as GNU/Linux). Contrary to where it is declared, the *defer* statement is only called after all the statements in the surrounding function, which makes sure no file descriptors (FDs) are leaked and the file is properly closed when the function returns.

In the next step at line 25 a database schema creation is attempted, handling the potential error in a Go idiomatic way, which uses the return value from the function in an assignment to a variable declared in the `if` statement, and checks whether the `err` was `nil` or not. In case the `err` was not `nil`, i.e. *there was an error in the callee function*, the condition evaluates to `true`, which is followed by entering the inner block. Inside it, the error is announced to the user (likely a developer running the test in this case) and the testing object's `FailNow()` method is called. That marks the test function as having failed, and thus stops its execution. In this case, that is the desired outcome, since if the database schema creation call fails, there really is no point in continuing the testing of user creation.

Conversely, if the schema *does* get created without an error, the code continues to declare a few variables (lines 30-32): `username`, `email` and `ctx`, where the context injected with the logger is saved. Two of them are subsequently (line 33) passed into

the `UsernameExists` function, `ctx` being the first argument and the database pointer and `username` following, while the `email` variable is only used at a later stage (line 46). The point of declaring them together is to give a sense of relatedness. The error value returned from this function is again checked (line 33) and if everything goes well, the `usernameFound` boolean value is checked next at line 38.

```

1 // modules/user/user_test.go
2 package user
3
4 import (
5     "context"
6     "testing"
7
8     "git.dotya.ml/mirre-mt/pcmt/ent/enttest"
9     "git.dotya.ml/mirre-mt/pcmt/slogging"
10    _ "github.com/xiaqidun/entps"
11 )
12
13 func getCtx() context.Context {
14     l := slogging.Init(false)
15     ctx := context.WithValue(context.Background(), CtxKey{}, l)
16     return ctx
17 }
18
19 func TestUserExists(t *testing.T) {
20     connstr := "file:ent_tests?mode=memory&fk=1"
21     db := enttest.Open(t, "sqlite3", connstr)
22     defer db.Close()
23
24     if err := db.Schema.Create(context.Background()); err != nil {
25         t.Errorf("failed to create schema resources: %v", err)
26         t.FailNow()
27     }
28
29     username := "dude"
30     email := "dude@b.cc"
31     ctx := getCtx()
32
33     usernameFound, err := UsernameExists(ctx, db, username)
34     if err != nil {
35         t.Errorf("error checking for username %s existence: %q", username, err)
36     }
37
38     if usernameFound {
39         t.Errorf("unexpected: user%s should not have been found", username)
40     }
41
42     if _, err := EmailExists(ctx, db, email); err != nil {
43         t.Errorf("unexpected: user email '%s' should not have been found", email)
44     }
45
46     usr, err := CreateUser(ctx, db, email, username, "so strong")
47     if err != nil {
48         t.Errorf("failed to create user, error: %q", err)
49         t.FailNow()
50     } else if usr == nil {
51         t.Error("got nil usr back")
52         t.FailNow()
53     }
54
55     if usr.Username != username {
56         t.Errorf("got back wrong username, want: %s, got: %s",
57             username, usr.Username,
58         )
59     } // ...more checks...
60 }

```

Listing 13.1 User existence integration test

Since the database has just been created, there should be no users, which is checked in the body of the `if` statement (line 35). The same check is then performed using an

email (line 42), which is also correctly expected to fail.

The final statements of the described test attempts to create a user by calling the function `CreateUser(...)` at line 46, whose return values are again checked for both error and *nilability*, respectively. The test continues with more of the checks similar to what has been described so far, but the rest was omitted for brevity.

As was just demonstrated in the test, a neat thing about error handling in Go is that it allows for very easy checking of all code paths, not just the *happy path* where there are no issues. The recommended approach of immediately explicitly handling (or deciding to ignore) the error is in author's view superior to wrapping hundreds of lines in `try` blocks and then *catching* (or not) *all the* exceptions, as is the practice in some other languages.

13.3 Test environment

The application has been deployed in a test environment on author's modest Virtual Private Server (VPS) at <https://testpcmt.dotya.ml>, protected by *Let's Encrypt* issued, short-lived, ECDSA `secp384r1` curve TLS certificate, and configured with strict CSP. It is a test instance, therefore limits (and rate-limits) to prevent abuse might be imposed.

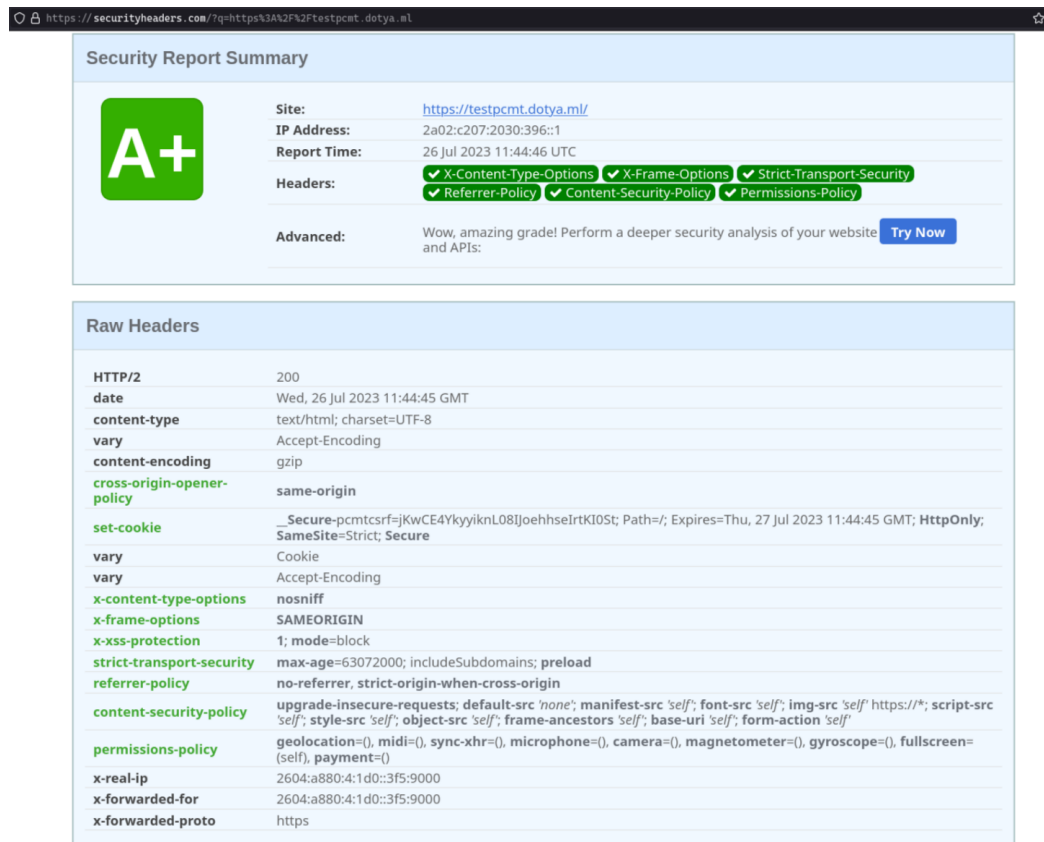
The test environment makes the program available over both modern IPv6 and legacy IPv4 protocols, to maximise accessibility. Redirects were set up from plain HTTP to HTTPS, as well as from `www` to non-`www` domain. The subject domain configuration is hardened by setting the CAA record, limiting certificate authorities (CAs) that are able to issue TLS certificates for it (and let them be trusted by validating clients). Additionally, *HTTP Strict Transport Security* (HSTS) had been enabled for the main domain (`dotya.ml`) including the subdomains quite some time ago (consult the preload lists in Firefox/Chrome), which mandates that clients speaking HTTP only ever connect to it (and the subdomains) using TLS.

13.3.1 Deployment validation

The deployed application has been validated using the *Security Headers* tool (see <https://securityheaders.com/?q=https%3A%2F%2Ftestpcmt.dotya.ml>), the results of which can be seen in Figure 13.1.

It shows that the application sets the Cross Origin Opener Policy to `same-origin`,

which isolates the browsing context exclusively to *same-origin* documents, preventing *cross-origin* documents from loading in the same browser context.



The screenshot displays a security report for the website `https://testpcmt.dotya.ml/`. The report shows an overall grade of **A+**. The site's IP address is `2a02:c207:2030:396::1` and the report was generated on `26 Jul 2023 11:44:46 UTC`. The headers section lists several security headers with green checkmarks indicating they are present: `X-Content-Type-Options`, `X-Frame-Options`, `Strict-Transport-Security`, `Referrer-Policy`, `Content-Security-Policy`, and `Permissions-Policy`. An advanced section suggests performing a deeper security analysis.

The raw headers section contains the following data:

Header	Value
HTTP/2	200
date	Wed, 26 Jul 2023 11:44:45 GMT
content-type	text/html; charset=UTF-8
vary	Accept-Encoding
content-encoding	gzip
cross-origin-opener-policy	same-origin
set-cookie	__Secure-pcmctsrfr=jKwCE4YkyiknL08JjoehhseIrtK10St; Path=/; Expires=Thu, 27 Jul 2023 11:44:45 GMT; HttpOnly; SameSite=Strict; Secure
vary	Cookie
vary	Accept-Encoding
x-content-type-options	nosniff
x-frame-options	SAMEORIGIN
x-xss-protection	1; mode=block
strict-transport-security	max-age=63072000; includeSubdomains; preload
referrer-policy	no-referrer, strict-origin-when-cross-origin
content-security-policy	upgrade-insecure-requests; default-src 'none'; manifest-src 'self'; font-src 'self'; img-src 'self' https://*; script-src 'self'; style-src 'self'; object-src 'self'; frame-ancestors 'self'; base-uri 'self'; form-action 'self'
permissions-policy	geolocation=(), midi=(), sync-xhr=(), microphone=(), camera=(), magnetometer=(), gyroscope=(), fullscreen=(self), payment=()
x-real-ip	2604:a880:4:1d0::3f5:9000
x-forwarded-for	2604:a880:4:1d0::3f5:9000
x-forwarded-proto	https

Figure 13.1 Security Headers scan

Furthermore, a Content Security Policy of `upgrade-insecure-requests; default-src 'none'; manifest-src 'self'; font-src 'self'; img-src 'self' https://*; script-src 'self'; style-src 'self'; object-src 'self'; form-action 'self'; frame-ancestors 'self'; base-uri 'self'` is set by the program using a header. This policy essentially pronounces the application (whatever domain it happens to be hosted at - 'self') as the only *permissible* source for any scripts, styles and frames, the only destination of web forms. One exception is the `img-src 'self' https://*` directive, which more leniently also permits images from any *secure* sources. This measure ensures that no unvetted content is ever loaded from elsewhere.

The Referrer-Policy header setting of `no-referrer, strict-origin-when-cross-origin` ensures that user tracking is reduced, since no referrer is included (the `Referer` header is omitted) when the user navigates away from the site or somehow send requests outside the application using other means. The Permissions-Policy set to `geolocation=(), midi=(), sync-xhr=(), microphone=(), camera=(), gyroscope=(), magnetometer=(), fullscreen=(self), payment=()` declares that the application is,

for instance, never going to request access to payment information, user microphone or camera devices, or geolocation.

`gobuster` was used in fuzzing mode to aid in uncovering potential application misconfigurations. The wordlists used include:

- Anton Lopanitsyn's `fuzz.txt` (<https://github.com/Bo0oM/fuzz.txt/tree/master>)
- Daniel Miessler's `SecLists` (<https://github.com/danielmiessler/SecLists>)
- Sam's `samlists` (<https://github.com/the-xentropy/samlists>)

Many requests yielded 404s for non-existent pages, or possibly pages requiring authentication (`NotFound` is used so as not to disclose page's existence). The program initially also issued quite a few 503s as a result of rate-limiting, until `gobuster` was tamed using the `--delay` parameter. Anti-CSRF measures employed by the program caused most of the requests to yield 400s (missing CSRF token), or 403s with a CSRF token.

The deployed application was scanned with Quallys' *SSL Labs* scanner and the results can be seen in Figure 13.2, confirming that HSTS (includes subdomains) is deployed, the server runs TLS 1.3, the DNS Certificate Authority Authorisation (CAA) is configured for the domain, with the overall grade being A+.

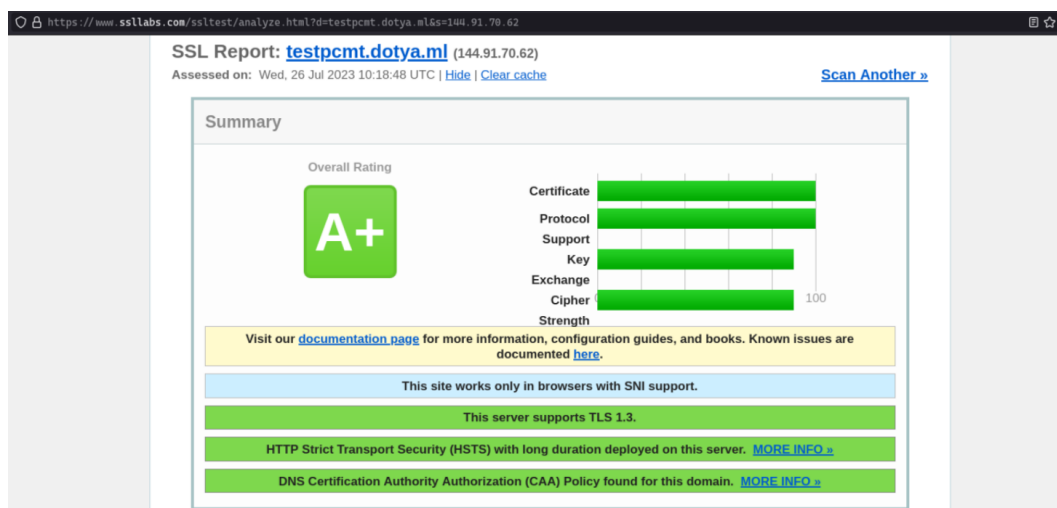


Figure 13.2 Quallys SSL Labs scan

14 APPLICATION SCREENSHOTS

Figure 14.1 depicts the initial page that a logged-out user is greeted with when they load the application.

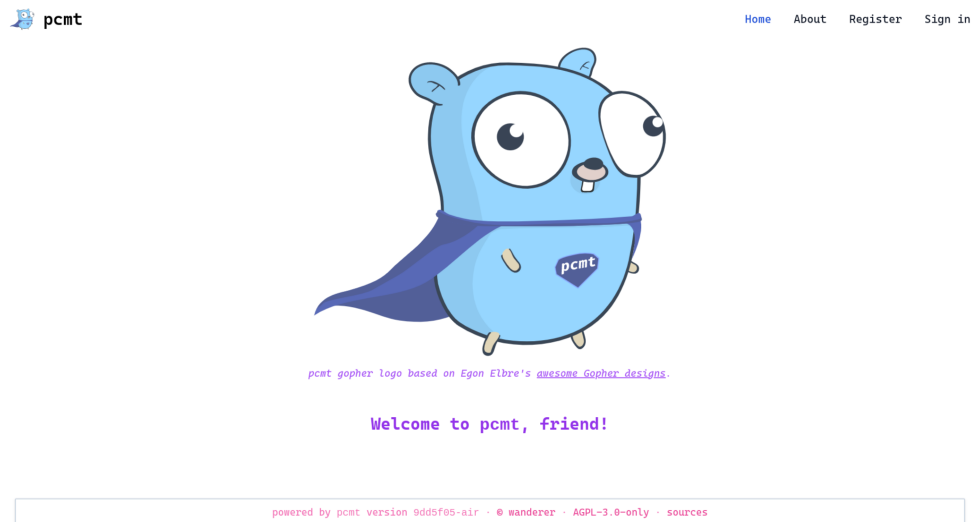


Figure 14.1 Homepage

Figure 14.2 is showing a registration page with input fields turned green after basic validation. Visiting this page with registration disabled in settings would yield a 404.

Welcome!

Create A New Account

[Already have an account?](#)

Figure 14.2 Registration page

Welcome!

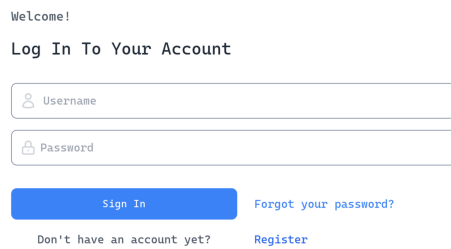
Create A New Account

Please provide a valid email address.

[Already have an account?](#)

Figure 14.3 Registration page
email error

A sign-up form error telling the user to provide a valid email address is shown in Figure 14.3.



Welcome!

Log In To Your Account

Username

Password

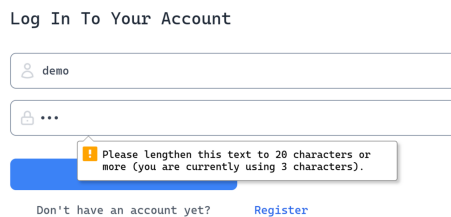
Sign In

Forgot your password?

Don't have an account yet? Register

Figure 14.4 Sign-in page

Figure 14.4 depicts a sign-in form similar to the sign-up one.



Log In To Your Account

demo

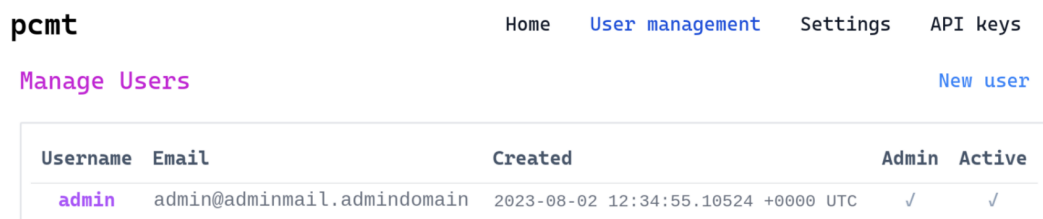
...

Please lengthen this text to 20 characters or more (you are currently using 3 characters).

Don't have an account yet? Register

Figure 14.5 Short password error on sign-in

An error in Figure 14.5 prompts the user to lengthen the content of the password field from 3 to at least 20 characters.



pcmt

Home User management Settings API keys

Manage Users New user

Username	Email	Created	Admin	Active
admin	admin@adminmail.admindomain	2023-08-02 12:34:55.10524 +0000 UTC	✓	✓

Figure 14.6 User management screen

Figure 14.6 shows the user management screen, which provides links to view user details page, start creating a new user.

User creation form can be seen in Figure 14.7. Both regular and admin level users can be created here. In this case, an error is shown, telling the user there is an issue with username uniqueness. User experience of this process could in the future be improved by using a bit of JavaScript (or WebAssembly) to check uniqueness of the username on user's *key-up*.

Create New User

Error: Username Is Not Unique

admin

a@b.cc

...

...

Is admin? Is active?

Create

Figure 14.7 User creation: 'username not unique' error

Successfully created user "demo"!

Manage Users New user

Username	Email	Created	Admin	Active
admin	admin@adminmail.admindomain	2023-08-02 12:34:55.10524 +0000 UTC	✓	✓
demo	a@b.cc	2023-08-07 16:12:07.056267 +0000 UTC	✗	✓

Figure 14.8 'demo' user creation post-hoc

The user management screen is again shown in Figure 14.8 after user 'demo' was created. An informative *flash* message is printed near the top of the page immediately after the action and not shown on subsequent page loads.

User Details ← All users

Edit Delete

ID:	4a8960c5-ced7-4b21-ac17-478f55052e7d
Username:	admin
Email:	admin@adminmail.admindomain
Admin:	true
Active:	true

Figure 14.9 User details screen

The user details page is depicted in Figure 14.9. The interface presents key information about the user such as ID, username and admin status. Additionally, it provides a link back to the previous page and two buttons: one for editing the user and one for user deletion.

Figure 14.10 User edit screen

Figure 14.10 shows the form for user editing with a button ‘Update’ in the bottom for submitting, a couple of checkboxes for toggling ‘admin’ and ‘active’ state of the user. Above those, there are input fields for ‘username’, ‘email’, ‘password’ and the confirmation of the password.

Figure 14.11 User deletion confirmation

When attempting to delete a user, the administrator is presented with the screen shown in Figure 14.11, which asks them whether they are absolutely sure to perform an action with permanent consequences. The ‘Confirm permanent deletion’ button is highlighted in intense red colour, while the ‘Cancel’ button is displayed in a light blue tone. There are two additional links: the ‘All users’ one that points to the user management page, and the ‘Back to detail’ one that simply brings the administrator one step back to the user details page.

Successfully deleted user "demo"!

Manage Users New user

Username	Email	Created	Admin	Active
admin	admin@adminmail.admindomain	2023-08-02 12:34:55.10524 +0000 UTC	✓	✓

Figure 14.12 User deletion post-hoc

After successful user deletion, the administrator is redirected back to user management page and a flash message confirming the deletion is printed near the top of the page, as shown in Figure 14.12.

Manage Your API Keys

Have I Been Pwned?

HIBP API key

Adds support for querying Troy Hunt's [Have I Been Pwned? API](#).

DeHashed.com

DeHashed.com API key

Adds support for querying [DeHashed.com API](#).

Figure 14.13 Manage API keys

Figure 14.13 shows a page that allows administrators to manage instance-wide API keys for external services, such as *Have I Been Pwned?* or *DeHashed.com*. Do note that these keys are never distributed to clients in any way and are only ever used by the application itself to make the requests on *behalf* of the users.

```

~/pcmt development +434 -20 v1.20.6 impure (devenv-shell)
→ ./pcmt -import ./modules/localbreach/testdata/test1.yaml
{"time":"2023-08-20T06:15:24.352206964+02:00","level":"INFO","msg":"sl
og logger initialised"}
{"time":"2023-08-20T06:15:24.671182294+02:00","level":"INFO","msg":"co
nnecting to db at 'host=127.0.0.1 sslmode=disable port=5432 user=postgres
dbname=postgres password=postgres'", "pcmt extra":{"module":"run"}}
{"time":"2023-08-20T06:15:24.671396891+02:00","level":"INFO","msg":"en
suring the db is set up and attempting to automatically migrate db sch
ema", "pcmt extra":{"module":"run"}}
{"time":"2023-08-20T06:15:24.791830227+02:00","level":"INFO","msg":"Im
port succeeded: 5[{Name:test breach 1 Description: CreatedAt:0001-01-0
1 00:00:00 +0000 UTC Date:2022-11-11 00:00:00 +0000 UTC IsVerified:fa
lse ContainsUsernames:false ContainsEmails:true ContainsPasswords:true
ContainsHashes:false HashType: HashSalted:false HashPeppered:false Dat
a:{Usernames:<nil> Emails:0xc000b02cd8 Hashes:<nil> Passwords:<nil>}]
{Name:test breach 2 Description: CreatedAt:0001-01-01 00:00:00 +0000 U
TC Date:2022-11-12 00:00:00 +0000 UTC IsVerified:false ContainsUsernam
es:false ContainsEmails:true ContainsPasswords:true ContainsHashes:fa
lse HashType: HashSalted:false HashPeppered:false Data:{Usernames:<nil>
Emails:0xc000b02cd8 Hashes:<nil> Passwords:<nil>}]", "pcmt extra":{"m
odule":"run"}}
✓ Import succeeded

~/pcmt development +434 -20 v1.20.6 impure (devenv-shell)
→
postgres@127:postgres> \du -x -v local_breaches;
SELECT 0
Time: 0.008s
postgres@127:postgres> \du -x -v local_breaches;
-[ RECORD 1 ]
id          | b3d43219-d091-43a1-94b3-1d1666ee0825
name       | test breach 1
date       | 2022-11-11 00:00:00+00
description |
is_verified | False
contains_passwords | True
contains_hashes | False
contains_emails | True
contains_usernames | False
hash_type   |
hash_salt   | False
hash_peppered | False
data       | {"Emails": ["aa@bb.cc", "dfjsnlads@gmail.com"], "H
ashes": null, "Passwords": null, "Usernames": null}
created_at  | 2023-08-20 04:15:24.33851+00
-[ RECORD 2 ]
id          | fe06633a-6f79-46bd-b70b-85b9732e71a3
name       | test breach 2
date       | 2022-11-12 00:00:00+00
description |
is_verified | False
contains_passwords | True
contains_hashes | False
contains_emails | True
contains_usernames | False
hash_type   |
hash_salt   | False
hash_peppered | False
data       | {"Emails": ["aa@bb.cc", "dfjsnlads@gmail.com"], "H
ashes": null, "Passwords": null, "Usernames": null}
created_at  | 2023-08-20 04:15:24.33851+00
SELECT 2
Time: 0.012s
postgres@127:postgres>

```

Figure 14.14 Import of locally available breach data from the CLI

Figure 14.14 depicts how formatted breach data can be imported into the program's database using the CLI.

Figure 14.15 displays the result of a search using the online data source. The account was not found to be a part of any of the available breaches.

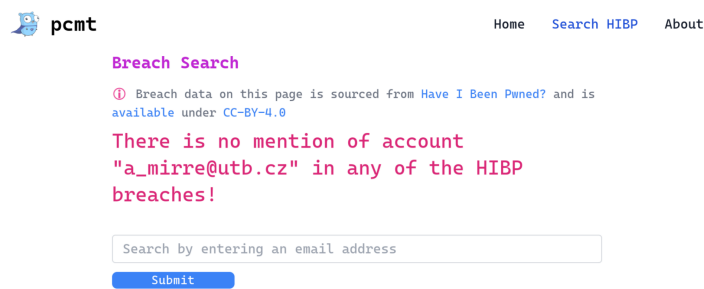


Figure 14.15 Compromise monitoring using online sources - no breach found

Figure 14.16 depicts the result of a search using the online API, providing a message and list of links to breach details.

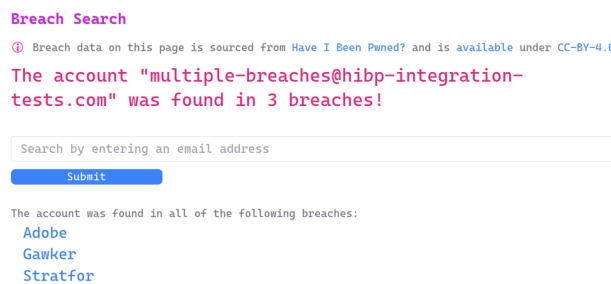


Figure 14.16 Compromise listing using online API (and a test account)



Figure 14.17 Stratfor breach details page

Figure 14.17 shows the *Stratfor* breach details page, with the data sourced from an online API.

CONCLUSION

The objectives of the thesis have been to create a tool that would enable users to verify the potentiality of their compromise in time, i.e. monitor it, by validating the assumptions on the security of their credentials.

In the theoretical part, conceptual foundations and technical underpinnings of common pieces of the infrastructure were attended to and explained, with a focus relating to creating web applications. Additionally, security mechanisms such as Site Isolation and Content Security Policy, commonly employed by mainstream browsers of today, were briefly introduced and it was proven how Content Security Policy could be configured simply and quickly. Furthermore, the criteria for local and online data sources were evaluated.

An extensive body of the thesis then revolved around the practical part, describing everything from tooling and development processes used, to high-level view of application architecture, and then dove into implementation details of specific parts of the application across the stack. Import of local breach data and constructing database queries using a graph-like API were also highlighted.

Various deployment and configuration scenarios were considered, the validation methods used to verify the correct working of the application were described and justified, and the practical part concluded by showing screenshots of the application in use.

The list of potential improvements for the future may also be amended by adding *fuzzing* tests for the program to help uncover potential bugs, producing Software Bill of Materials to aid in ensuring compliance, and utilising additional immutable database for activity logs.

The program does have a very solid core, it listens for OS signals and can handle shutdowns gracefully. It supports structured logging, with the option to plug in a log exporter. Most importantly, it gives users a tool in the battle against the always vigilant attackers that are after their passwords.

Even though it might not be called an utterly *finished* project yet, it can already serve a clear purpose.

REFERENCES

- [1] Wang, X.; Yu, H.: How to Break MD5 and Other Hash Functions. 05 2005, ISBN 978-3-540-25910-7, pp. 561–561, doi:10.1007/11426639_2.
- [2] Klíma, V.: Tunnels in Hash Functions: MD5 Collisions Within a Minute. *IACR Cryptology ePrint Archive*, volume 2006, January 2006: p. 105.
- [3] Sasaki, Y.; Aoki, K.: *Finding Preimages in Full MD5 Faster Than Exhaustive Search*. Springer, Berlin, Heidelberg, 2009, pp. 134–152, doi:10.1007/978-3-642-01001-9_8.
- [4] Mao, M.; Chen, S.; Xu, J.: Construction of the Initial Structure for Preimage Attack of MD5. In *2009 International Conference on Computational Intelligence and Security*, volume 1, 2009, pp. 442–445, doi:10.1109/CIS.2009.214.
- [5] O’Connor, J.; Aumasson, J.-P.; Neves, S.; et al.: BLAKE3 - one function, fast everywhere. [online], 2021, Available from: <https://raw.githubusercontent.com/BLAKE3-team/BLAKE3-specs/master/blake3.pdf> [viewed 2023-08-14].
- [6] Goodin, D.: Why passwords have never been weaker—and crackers have never been stronger. [online], August 2012, Available from: <https://arstechnica.com/information-technology/2012/08/passwords-under-assault/> [viewed 2023-08-13].
- [7] Thorsheim, P.: LinkedIn Password Infographic. [online], June 2012, Available from: <https://securitynirvana.blogspot.com/2012/06/linkedin-password-infographic.html> [viewed 2023-08-13].
- [8] m3g9tr0n: Cracking Story - How I Cracked Over 122 Million SHA1 and MD5 Hashed Passwords. [online], 2012, Available from: <https://blog.thireus.com/cracking-story-how-i-cracked-over-122-million-sha1-and-md5-hashed-passwords/> [viewed 2023-08-13].
- [9] Velazco, C.: 6.5 Million LinkedIn Passwords Reportedly Leaked, LinkedIn Is “Looking Into” It. [online], 2012, Available from: <https://techcrunch.com/2012/06/06/6-5-million-linkedin-passwords-reportedly-leaked-linkedin-is-looking-into-it/> [viewed 2023-08-13].
- [10] Perez, S.: 117 million LinkedIn emails and passwords from a 2012 hack just got posted online. [online], May 2016, Available from: <https://techcrunch.com/2016/05/18/117-million-linkedin-emails-and-passwords-from-a-2012-hack-just-got-posted-online/> [viewed 2023-08-13].

-
- [11] Imperva: Consumer Password Worst Practices. [online], 2014, Available from: https://www.imperva.com/docs/gated/WP_Consumer_Password_Worst_Practices.pdf [viewed 2023-08-13].
- [12] Goodin, D.: 13 million plaintext passwords belonging to webhost users leaked online. [online], 2015, Available from: <https://arstechnica.com/information-technology/2015/10/13-million-plaintext-passwords-belonging-to-webhost-users-leaked-online/> [viewed 2023-08-13].
- [13] Forcepoint: Chinese Internet Suffers the Most Serious User Data Leak in History. [online], December 2011, Available from: <https://www.forcepoint.com/blog/x-labs/chinese-internet-suffers-most-serious-user-data-leak-history> [viewed 2023-08-13].
- [14] Goodin, D.: 6.6 million plaintext passwords exposed as site gets hacked to the bone. [online], September 2016, Available from: <https://arstechnica.com/information-technology/2016/09/plaintext-passwords-and-wealth-of-other-data-for-6-6-million-people-go-public/> [viewed 2023-08-13].
- [15] Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018, doi:10.17487/RFC8446, Also available from: <https://www.rfc-editor.org/info/rfc8446>.
- [16] Alexander, K.: Speakeasies of the Prohibition Era. [online], December 2022, Available from: <https://www.legendsofamerica.com/ah-prohibitionspeakeasy/> [viewed 2023-05-24].
- [17] National Institute of Standards and Technology: Passphrase. [online], Available from: <https://csrc.nist.gov/glossary/term/Passphrase> [viewed 2023-05-24].
- [18] McMillan, R.: The World's First Computer Password? It Was Useless Too. [online], January 2012, Available from: <https://www.wired.com/2012/01/computer-password/> [viewed 2023-05-24].
- [19] Lars Klint: Excuse me @EtihadAirways, why do you insist on making my passwords worse? [online], June 2016, Available from: <https://twitter.com/larsklint/status/748615185762484224> [viewed 2023-05-24].
- [20] Etihad Airways: Reply to Lars Klint. [online], June 2016, Available from: <https://twitter.com/EtihadAirways/status/748626413306150912> [viewed 2023-05-24].

- [21] Nick Heer: This does not give me confidence in your password security, @YourAlberta. (cc. @troyhunt). [online], July 2017, Available from: <https://twitter.com/nickheer/status/887196833872658432> [viewed 2023-05-24].
- [22] Cerf, V.: ASCII format for network interchange. RFC 20, October 1969, doi:10.17487/RFC0020, Also available from <https://www.rfc-editor.org/info/rfc20>.
- [23] ISO/IEC 10646:2020: Information technology – Universal Coded Character Set (UCS). Standard, International Organization for Standardization, Geneva, CH, 2020.
- [24] National Cyber Security Centre: Password policy: updating your approach. [online], November 2018, Available from: <https://twitter.com/nickheer/status/887196833872658432> [viewed 2023-05-24].
- [25] The Chromium Projects: Chromium Security – Site Isolation. [online], 2023, Available from: <https://www.chromium.org/Home/chromium-security/site-isolation/> [viewed 2023-05-24].
- [26] Anny Gakhokidze: Introducing Firefox’s new Site Isolation Security Architecture. [online], May 2021, Available from: <https://hacks.mozilla.org/2021/05/introducing-firefox-new-site-isolation-security-architecture/> [viewed 2023-05-24].
- [27] The Open Worldwide Application Security Project: OWASP Top 10:2021. [online], 2021, Available from: <https://owasp.org/Top10/> [viewed 2023-05-24].
- [28] Weichselbaum, L.; Spagnuolo, M.; Janc, A.; et al.: CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy. In *Proceedings of the 5th ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*, New York, NY, USA, 2016, ISBN 978-1-4503-4139-4/16, pp. 1376–1387, doi:10.1145/297674.
- [29] Stamm, S.; Sterne, B.; Markham, G.: Reining in the Web with Content Security Policy. In *Proceedings of the 19th International Conference on World Wide Web*, Raleigh, North Carolina, USA, 2010, ISBN 978-1-60558-799, pp. 921–930.
- [30] NixOS Contributors: Nix Language. [online], Nix Reference Manual, 2023, Available from: <https://nixos.org/manual/nix/stable/language/index.html>. [viewed 2023-05-17].
- [31] NixOS Contributors: How Nix Works. [online], 2023, Available from: <https://nixos.org/guides/how-nix-works.html>. [viewed 2023-05-17].

- [32] The Dhall Language Contributors: Dhall Configuration Language. [online], 2017, Available from: <https://dhall-lang.org>. [viewed 2023-05-17].
- [33] The Dhall Language Contributors: Safety Guarantees. [online], 2023, Available from: <https://docs.dhall-lang.org/discussions/Safety-guarantees.html?highlight=normalization>. [viewed 2023-05-18].
- [34] Bastian, W.; Karlitskaya, A.; Poettering, L.; et al.: XDG Base Directory Specification. [online], May 2021, Available from: <https://specifications.freedesktop.org/basedir-spec/basedir-spec-latest.html>. [viewed 2023-05-17].
- [35] Troy Hunt: The Have I Been Pwned API Now Has Different Rate Limits and Annual Billing. [online], 2022, Available from: <https://www.troyhunt.com/the-have-i-been-pwned-api-now-has-different-rate-limits-and-annual-billing/> [viewed 2023-08-15].
- [36] The PostgreSQL Global Development Group: Postgres 15 Authentication Methods. [online], 2023, Available from: <https://www.postgresql.org/docs/15/auth-methods.html> [viewed 2023-05-17].
- [37] Hansen, T.: SCRAM-SHA-256 and SCRAM-SHA-256-PLUS Simple Authentication and Security Layer (SASL) Mechanisms. RFC 7677, November 2015, doi:10.17487/RFC7677, Also available from <https://www.rfc-editor.org/info/rfc7677>.
- [38] Taishi Kasuga: Password-encryption tool for PostgreSQL with SCRAM-SHA-256. [online], 2023, Available from: <https://github.com/supercaracal/scram-sha-256>. [viewed 2023-05-17].
- [39] Ayer, A.: It's Now Possible To Sign Arbitrary Data With Your SSH Keys. [online], November 2021, Available from: https://www.agwa.name/blog/post/ssh_signatures. [viewed 2023-05-17].
- [40] Harmon, E.; Stoltz, M.: GitHub Reinstates youtube-dl After RIAA's Abuse of the DMCA. [online], November 2020, Available from: <https://www.eff.org/deeplinks/2020/11/github-reinstates-youtube-dl-after-riaas-abuse-dmca>. [viewed 2023-05-24].
- [41] Valsorda, F.; Cox, B.; age contributors: A simple, modern and secure encryption tool (and Go library) with small explicit keys, no config options, and UNIX-style composability. [online], 2021, Available from: <https://github.com/FiloSottile/age>. [viewed 2023-05-23].

-
- [42] Langley, A.; Hamburg, M.; Turner, S.: Elliptic Curves for Security. RFC 7748, January 2016, doi:10.17487/RFC7748, Also available from <https://www.rfc-editor.org/info/rfc7748>.
- [43] Digital Forensics & Computer Security Research: LiME - Linux Memory Extractor. [online], 2007, Available from: <https://github.com/504ensicsLabs/LiME>. [viewed 2023-05-23].
- [44] The Dhall Language Contributors: Prelude-v23.0.0. [online], 2023, Available from: <https://store.dhall-lang.org/Prelude-v23.0.0/>. [viewed 2023-05-24].
- [45] Brent Baude: Podman: Managing pods and containers in a local container runtime. [online], May 2019, Available from: <https://developers.redhat.com/blog/2019/01/15/podman-managing-containers-pods> [viewed 2023-07-24].
- [46] Howe, D.: Linux from FOLDOC. [online], 2000, Available from: <https://foldoc.org/linux>. [viewed 2023-05-17].
- [47] Stallman, R.: Linux and the GNU System. [online], November 2021, Available from: <https://www.gnu.org/gnu/linux-and-gnu.html>. [viewed 2023-05-17].

LIST OF ABBREVIATIONS

SHA	Secure Hash Algorithm
AES	Advanced Encryption Standard
CSPRNG	Cryptographically Secure Pseudo-Random Number Generator
ID	Identity
PID	Process ID
Cgroup	Control group
TLS	Transport Layer Security
TCP	Transmission Control Protocol
SSH	Secure Shell
DNS	Domain Name System
ZSTD	Z Standard
ZFS	Zettabyte File System
ISP	Internet Service Provider
GPG	GNU Privacy Guard
GNU	GNU's Not Unix!
CSS	Cascading Style Sheets
API	Application Programming Interface
CLI	Command Line Interface
SCM	Source Code Management
HIBP	Have I Been Pwned?
TDD	Test Driven Development
TOML	Tom's Obvious Minimal Language
YAML	Yet Another Markup Language
JSON	Java Script Object Notation
INI	Initialization file
CPU	Central Processing Unit
RAM	Random Access Memory
NVMe	Non-Volatile Memory Express
PCIe	Peripheral Component Interconnect Express
HPC	High Performance Computing
OOM	Out of Memory
OWASP	Open Web Application Security Project
NIST	National Institute of Standards and Technology
SEO	Search Engine Optimisation

LIST OF FIGURES

Fig. 2.1.	Short arbitrary password length limit.....	17
Fig. 2.2.	Forbidden special characters in passwords	18
Fig. 9.1.	Drone CI median build time	37
Fig. 10.1.	Application class diagram.....	40
Fig. 10.2.	Application use case diagram	46
Fig. 13.1.	Security Headers scan	64
Fig. 13.2.	Quallys SSL Labs scan	65
Fig. 14.1.	Homepage.....	66
Fig. 14.2.	Registration page	66
Fig. 14.3.	Registration page email error	66
Fig. 14.4.	Sign-in page.....	67
Fig. 14.5.	Short password error on sign-in.....	67
Fig. 14.6.	User management screen	67
Fig. 14.7.	User creation: 'username not unique' error	68
Fig. 14.8.	'demo' user creation post-hoc.....	68
Fig. 14.9.	User details screen.....	68
Fig. 14.10.	User edit screen.....	69
Fig. 14.11.	User deletion confirmation	69
Fig. 14.12.	User deletion post-hoc.....	69
Fig. 14.13.	Manage API keys	70
Fig. 14.14.	Import of locally available breach data from the CLI	70
Fig. 14.15.	Compromise monitoring using online sources - no breach found.....	71
Fig. 14.16.	Compromise listing using online API (and a test account).....	71
Fig. 14.17.	Stratfor breach details page	71

LIST OF TABLES

Tab. 9.1.	Tool/Library-Usage Matrix.....	39
Tab. 9.2.	Dependency-Version Matrix.....	39

LIST OF LISTINGS

List. 1.1	Broken collision resistance of MD5	14
List. 5.1	Breach ImportSchema Go struct (imports from the standard library assumed)	27
List. 5.2	A YAML file containing breach data formatted according to the ImportSchema, optionally containing multiple documents.....	28
List. 9.1	Verifying the signature of a git commit	35
List. 9.2	Prepare allowed signers file and signature format for git	36
List. 10.1	Example formatting expression supplied to the logger	42
List. 10.2	Conditionally enabling functionality inside a Go template based on user access level.....	44
List. 11.1	Dhall configuration schema version 0.0.1-rc.2	49
List. 11.2	Dhall configuration defaults for schema version 0.0.1-rc.2	51
List. 11.3	Ent graph query	54
List. 12.1	Example application deployment using rootless Podman	56
List. 12.2	Pinging pod containers using their names	57
List. 12.3	Podman pod port binding inspection	58
List. 12.4	In-pod database is unreachable from the host.....	58
List. 12.5	Example reverse proxy configuration snippet.....	59
List. 13.1	User existence integration test.....	62

LIST OF APPENDICES

- A I. List of supplemental material
- A II. Whys
- A III. Terminology

APPENDIX A I. LIST OF SUPPLEMENTAL MATERIAL

1.1 Git signing keys

File name: `gitsign.pub`

Blake3: `af0f319aecdc3ca1d1d4002ebd4ebd5f93e6030551a91f3e09664995340208c85`

SHA3-256: `8cab97c322c26369437ba365159b14ac4b829056c8974c671fdc57bcba8f518b`

File name: `surtur.pub`

Blake3: `f6d87e457c9939063117a2ee243af279b39b07f47bbb190f3fe5b59a6972c531`

SHA3-256: `0b53a5766c3f144fffa8ff7c1c24c5f10a52588cd75d86fb8150aacbbd7c7b10`

1.2 ZSTD-compressed tarball of the PCMT source code repository

Note: Also contains the git history.

File name: `pcmt.git.tar.zst`

Blake3: `ca53c60eb4f0f14e287ec866ccba63ef7e375c17c1fbbcd58950162f7d8ad172`

SHA3-256: `3df969231ac8b464d9ed6ea6991aa61751629b997b4928cece2c5261543c3803`

1.3 ZSTD-compressed tarball of the PCMT configuration schema repository

Note: Also contains the git history.

File name: `pcmt-config-schema.tar.zst`

Blake3: `2f55e255318967e89443b5a1847466952599a5eae1080b8c973a0368920dbdcc`

SHA3-256: `9a0b383fce6bf2918fc39b44a5d29dccf27a77adc6c4b4483cd05e8be410345a`

1.4 ZSTD-compressed tarball of the L^AT_EX sources of this thesis

Note: Also contains the git history. The hash digest of this file is omitted to avoid chicken-egg problem when wishing to include its digest here.

File name: `pcmt-latex-thesis.tar.zst`

1.5 BLAKE3 checksums

Note: A list of BLAKE3 checksums for the supplementary material (excluding the checksum files). Check using `b3sum -c b3sums`.

File name: `b3sums`

Blake3: `2cbb654aff2755eb00a0d61ce68dabdb9348b8d2f283e95b7619d39ec451f729`

SHA3-256: `b0c26f0ee638e1e26e3dfbf2973e50a6130c446f5e9ed5db2aa3295e08890f93`

1.6 **SHA3-256** checksums

Note: A list of **SHA3-256** checksums for the supplementary material (excluding the checksum files). Check using `sha3-256sum -c sha3-256sums`.

File name: `sha3-256sums`

Blake3: 464d1be89b7d8ae62ff4687722127498f4d9b91b86f94b83fc7a07c97743916c

SHA3-256: cc57c0cfc5ddd9e4db754e914e4d87733b967d0040356108ebc225ac9cacf68e

APPENDIX A II. WHYS

This appendix is concerned with explaining why certain technologies were used.

2.1 Why Go

First, a question of ‘*Why pick Go for building a web application?*’ might arise, so the following few lines will try to address that.

Go¹⁾, or *Golang* for SEO-friendliness and disambiguating Go the ancient game, is a strongly typed, high-level *garbage-collected* language where functions are first-class citizens and errors are values.

The appeal for the author comes from a number of features of the language, such as built-in support for concurrency and unit testing, sane *zero* values, lack of pointer arithmetic, inheritance and implicit type conversions, easy-to-read syntax, producing a statically linked binary by default, etc., on top of that, the language has got a cute mascot. Thanks to the foresight of the Go Authors regarding *the formatting question* (i.e. where to put the braces, **tabs vs. spaces**, etc.), most of the discussions on this topic have been foregone. Every *gopher*²⁾ is expected to format their source code with the official formatter (`gofmt`), which automatically ensures that the code adheres to the one formatting standard. Then, there is *The Promise* of backwards³⁾ compatibility for Go 1.x, which makes it a good choice for long-term without the fear of being rug-pulled.

2.2 Why Nix/devenv

Nix (<https://builtwithnix.org/>) is a functional programming language resembling Haskell and a declarative package manager, which has been used in this project in the form of `devenv` tool (<https://devenv.sh/>) to create **declarable** and **reproducible** development environment. The author has previously used Nix directly with *flakes* and liked `devenv`, as it effectively exposed only a handful of parameters for configuration, and rid of the need to manage the full flake, which is of course still an option for people who choose so. See `devenv.nix` in the repository root.

¹⁾The Go programming language (<https://go.dev>)

²⁾euph. a person writing in the Go programming language

³⁾Now there is also the promise of *forward compatibility* (<https://go.dev/blog/toolchain>)

APPENDIX A III. TERMINOLOGY

3.1 Linux

The term *Linux* is exclusively used in the meaning of the Linux kernel [46].

3.2 GNU/Linux

As far as a Linux-based operating system is concerned, the term “GNU/Linux” as defined by the Free Software Foundation [47] is used. While it is longer and arguably a little bit cumbersome, the author aligns with the opinion that this term more correctly describes its actual target. Being aware that there are many people who conflate the complete operating system with its (be it core) component, the kernel, the author is taking care to distinguish the two, although writing from experience, colloquially, this probably brings more confusion and a lengthy explanation is usually required.

3.3 The program

By *the program* or *the application* without any additional context the author most probably means the Password Compromise Monitoring Tool program.