Secure Programming Assignment

Submitted for David Aspinall

June 18, 2024

Part A: Security-By-Construction?

1) CbyC proponents say usual development is ”construction-by-correction”. Discuss why this could be said for an immature SSDLC, giving two examples of correction in the process.

”Construction-by-correction” is succinctly defined by David Norfolk: ”build it wrong and fix the errors afterwards”[7]. In contrast to Correctness-By-Construction (CbyC), which Bordis et al. define as a ”technique for creating functionally correct programs guided by specification”[2]. Or, even more succinctly from David Norfolk: ”build it right in the first place”[7].

Two examples of corrections are provided to give some flesh to the contrast between these two approaches, and to spell out why CbyC represents progress in at least some contexts. Nb. The corrections provided are not obviously security related, but it is not strictly necessary for them to be. Even seemingly innocuously incorrect programs can become vulnerable in the hands of a sufficiently creative security researcher:

1. A large server side application is being written in Python. Unit testing reveals functions will fail to throw an error if arguments of the incorrect type are provided. This potentially means confusing errors surfacing far later than they ought to in the call stack. The functions must now be rewritten to improve their type checking so that issues are caught earlier and do not result in wider system failure.

   Finding type errors via unit test is dangerous from a system design standpoint, since it relies on developers *a)* writing tests that *b)* provide meaningful coverage against malformed arguments, something they are not guaranteed to do. Far better to write the application in a type-safe language, such as Golang, so issues of this kind are caught by the language compiler/interpreter's type-checker.

2. Integration testing reveals a client requesting data from a server does not receive the correct data due to API incompatibility. The client constructs a JSON blob to send as the body of an HTTP request, this request makes use of a field that the server does not recognise due to a typo. The server makes a best effort to satisfy the client, ignoring the invalid field, and returns results that satisfy the other filter fields that were passed that were valid.

   A good mitigation would be to define an API specification from which both the client and server code will be generated. The use of `.protobuf` files in the gRPC protocol[1] is a good example.

OWASP's Software Assurance Maturity Model (SAMM)[11] gives us a notion of what makes for an immature SSDLC. The SAMM defines 5 'business functions', each with a number of 'security practices'. In general it advocates ”shift-left” style policies. SAMM endorses practices that favour fixing security issues sooner, or systematically prevent them from coming into existence in the first place, regarding them as more mature. By contrast practices that apply security controls later to correct security issues post hoc are regarded as less mature. Two examples from SAMM of this pattern are provided below to make this assertion more concrete:

1. Within the Security Testing practice in the Verification business function a relatively immature (level 1) organisation is said to be conducting security testing, both manual and by means of automated tools. By contrast a relatively mature organisation is said to have embedded security testing as part of the development and deployment processes.

2. In the Security Architecture practice of the Design business function a less mature organisation is said to be inserting consideration of proactive security guidance into the software design process. By contrast a more mature organisation is said to be formally controlling the software design process and validating utilization of secure components. Nb. This maturity criterion in particular is the one that endorses the proposed CbyC-style SDLC improvements in the two correction examples above.

If a more mature organisation shifts left, resolves issues early, and prevents others from coming into existence by leveraging formal methods, then clearly a relatively mature organisation would favour CbyC where possible, and by contrast it would be indicative of a relatively immature organisation SSDLC to see "construction-by-correction" running rampant and CbyC falling by the wayside.

2) Another idea of CbyC is to "say things only once". Briefly discuss the pros and cons of this motto, applied to secure programming. Reference at least two secure software development lifecycle activities to support your discussion.

"Say things only once" is assumed to be synonymous with DRY (Don't Repeat Yourself). An advantage is that vulnerabilities found and fixed in one place will be fixed everywhere. Consider the following scenario: A company is introducing a new static analysis tool to their CI pipelines as part of a programme of improvements to their SSDLC. On its initial run it identifies an SQLi vulnerability in the authentication layer which is verified to be exploitable. The authentication layer is used in ∼100 internal services in line with the company's advocacy for DRY practices. A security engineer patches the issue, releases a new version of the software and issues an internal advisory that the version of the authentication layer needs to be updated by all services that consume it within a week. The patch is consumed in another week by 90% of teams, and 100% of the ones where the vulnerability was regarded as critical.

This could be regarded as a case against DRY, since the vulnerability being in one place meant that it was everywhere. On aggregate however it builds a case in favour of DRY. An alternative to this scenario is each of the ∼100 different services using their own authentication layer, resulting in roughly 30% of them being revealed to have SQLi vulnerabilities, and each in different ways. Five security engineers are now preoccupied for a full quarter submitting pull requests to patch 30 services they aren't familiar with.

Alternatively, consider a different scenario: The very same shared authentication layer accepts long-lived API bearer tokens for authentication. Tokens are cryptographically signed, with public keys passed as config items so that tokens can be verified without API calls to any other service. As part of the reformed SSDLC, external white-box penetration tests are being conducted. The first iteration of this discovers a token in source control. It is assumed to have leaked, so the security team looks for the token to be revoked. At this point the security team realises token revocation is not a feature supported by the authentication service. Developers maintaining the Single-Sign-On (SSO) service have requested this feature before, and now demand it with urgency. Developers maintaining the company's databases have historically resisted the feature, since it requires an API call every time the token is verified in order to check for revocation. An unacceptable performance penalty for their use case. The option of making token revocation configurable, such that it can be turned on or off, is discussed, but it is ultimately agreed that it would make the service so different that it makes more sense for the database layer to use a fork of the current authentication layer, and other services will be granted the feature.

Overzealous adherence to DRY prevented the revocation feature being implemented sooner. The two consumers use cases may have seemed similar, but were liable to diverge as the company's security maturity advanced. This inflexibility led to services having a vulnerability by design for longer than necessary. The more generic lesson is that DRY practices create dependencies between software services, which can make releasing new features and patches more cumbersome.

3) Consider a programming language which is type safe, such as Rust. Explain carefully what this means in practice. To what extent do you think we could say that programs written in Rust could have *Security by Construction*? What else, if anything, do you think would be needed for "SecbyC"?

In practice type safety means the compiler/interpreter implements features to prevent type errors, rather than leaving it to the developer. These features include but may not be limited to:

- Operators prevent logically confusing operations, for instance, adding a string to an integer.
- Functions have type signatures for both arguments and return values. Passing parameters or assigning the value returned to a variable with a type that differs from the signature will result in an error.
- Implicit type coercion is impossible or restricted. Eg. a variable that has been typed as an `int` cannot have a value assigned to it that has been previously typed as a `Long`, even though there may be a logically consistent way to do this. An explicit type cast must be made.
- A mitigation exists for the dangling pointer problem. Ie. if a language provides the ability to reference the address of an object A in memory, then there will exist a mechanism to prevent A being used after that address in memory has been freed, since that memory could now be in use for data that is not of the same type. Most type safe languages that allow references to addresses in memory achieve this with a garbage collector, memory is only freed when the last reference to it can no longer be used. Rust is the exception, making use of a borrow checker, which ensures that references do not outlive the data they reference.

Compared to other programming languages **Rust could be said to be relatively secure-by-construction**. Its type system satisfies all of the constraints listed above, mitigating their corresponding incorrect behaviours. It's ownership model, discussed in greater detail in Part B, elegantly prevents memory corruption and data races. Yet the statement that Rust is definitively Secure-By-Construction feels too strong. Rust programs can go wrong. Functions can panic, and the prevalence of unsafe tags, also discussed later, means undefined behaviour remains possible. It also feels strange to suggest that the correctness, and by extension security, of a program could ever be entirely guaranteed by a compiler.

In pursuit of greater Sec-By-C we ought to look to means of specifying the desired behaviour of a program, and verifying that the program satisfies those constraints via formal methods. For instance via the Coq theorem prover[4]. The rationale being that insecure programs are a subset of incorrect programs. If we can make programs correct via formal methods, then we can make them secure by extension.

Part B: Mechanisms in Rust

1) What kinds of programming vulnerability do you think can be reduced or eliminated using Rust's "move" semantics? Give three examples of different vulnerability kinds, comparing with code written in C or another language for contrast.

Thread Safety:

Consider the following C code. This represents the most succinct race condition I could write:

```c
#include <pthread.h>
#include <stdio.h>

void* increment(void *count) {
    int *count_int = (int*)count;
    for (int i = 0; i < 50000; i++) {
        *count_int = *count_int + 1;
    }
}
void main() {
    int count = 0;
    pthread_t thread1, thread2;
    pthread_create(&thread1, NULL, increment, &count);
    pthread_create(&thread2, NULL, increment, &count);
    pthread_join(thread1, NULL); pthread_join(thread2, NULL);
    printf("Count: %d\n", count);
}
```

Now consider a best effort equivalent in Rust (spoiler alert: this won't compile):

```rust
use std::thread;

fn increment(count: &mut String) {           // Nb. Shennanigans here of making count a
    for _ in 0..50000 {                       // String and casting back and forth made
        let n: i32 = count.parse().unwrap(); // necessary by i32 implementing the Copy
        *count = (n + 1).to_string();        // trait and therefore doing a pass by value
    }                                         // by default in spite of the move statement.
}                                             // String does not implement the copy trait.
fn main() {
    let mut count = String::from("0");
    let handle1 = thread::spawn(move || { increment(&mut count); });
    let handle2 = thread::spawn(move || { increment(&mut count); });
    handle1.join().unwrap(); handle2.join().unwrap();
    println!("Count: {}", count);
}
```

The C program's `increment` function iteratively reads the value of `*count`, adding `1` to it, and writing the result to `*count`. The intended behaviour is presumed to be that each thread executes this logic `50,000` times independently of one another, with a final result of a `*count` value of `100,000`. With both threads running these operations at the same time, the following order of operations becomes possible:

1. `thread1` reads the value of `*count` as `15` and calculates `15 + 1 = 16`
2. `thread2` reads the value of `*count` as `15` and calculates `15 + 1 = 16`
3. `thread1` writes a value of `16` to `*count`
4. `thread2` writes a value of `16` to `*count`

Both threads have tried to increment the value of `*count` by one, yet in real terms only one has succeeded. The end result is that the program non-deterministically prints a value in the range `50,000-100,000`.

It is impossible to duplicate this behaviour in Rust without `unsafe` tags. The key is the `move` statement. This forces the closure on line 11 to take ownership of `count`. Now `count` cannot be used outside the closure. This means a compilation error on line 12 (and 14): "`value used here after move`".

Race conditions occur when two or more processes running simultaneously expect a change to a shared state *not* to occur at a critical moment, when in fact it might. The code above is a toy example of the issue, but it demonstrates how Rust's move semantic and single owner requirement for mutable references eliminates this class of bugs. To see the security benefit consider Dirty COW[8], a race condition in copy-on-write functionality on pages in the Linux kernel. Had `mm/gup.c` been written in Rust, the resulting privilege escalation, actively exploited throughout October 2016, would not have been possible.

Use-After-Free (UAF):

Consider the following C. Its behaviour is undefined, however a possible result of running this (at least on my machine) is that the `printf` on line 9 will print `2`, because the space allocated, written into, and freed for `ptr1` is reused by `ptr2`.

```c
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      int *ptr1 = (int *)malloc(sizeof(int)); *ptr1 = 1;
6      printf("*ptr1: %d\n", *ptr1);
7      free(ptr1);
8      int *ptr2 = (int *)malloc(sizeof(int)); *ptr2 = 2;
9      printf("*ptr1 after free: %d\n", *ptr1);
10     return 0;
11 }
```

Programs like this become exploitable when the dangling pointer left behind can be read or written to, providing a user with access to memory they ought not have. Contrast this with the equivalent code in Rust:

```
1 fn main() {
2     let ptr1 = Box::new(1);
3     println!("*ptr1: {}", *ptr1);
4     drop(ptr1);
5     let ptr2 = Box::new(2);
6     println!("*ptr1 after free: {}", *ptr1);
7 }
```

Once again, this will not compile, because Rust considers the invocation of `drop` on line 4 to have moved `ptr1` out of scope, resulting in an error on line 6: `value borrowed here after move`. Rusts move semantic prevents the use of dangling pointers at compile time, and therefore eliminates another class of bug altogether.

To appreciate the potential impact consider CVE-2022-0609[9], a UAF in the Chrome browsers animation leading to RCE, exploited by North Korean backed threat actors[19]. In 2022 half of known exploitable bugs in Chrome were UAFs[16]. If browsers were written entirely in Rust this percentage ought to be approaching zero. All this with the caveat of the existence of this repo[15] which I have not fully understood.

Double Free:

`free()` in C is not idempotent. If called on a pointer twice it can result in the same block of memory appearing twice in the linked list that is maintained to keep track of free addresses on the heap. This can mean undefined behaviour, eg. subsequent calls to `malloc` may cause two different pointers to point to the same address in memory. The following C is lifted straight from OWASP's page on double frees[12]:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #define BUFSIZE1    512
5  #define BUFSIZE2    ((BUFSIZE1/2) - 8)  // Account for 8 bytes of metadata
6  int main(int argc, char **argv) {
7    char *buf1R1; char *buf2R1; char *buf1R2;
8    buf1R1 = (char *) malloc(BUFSIZE2);
9    buf2R1 = (char *) malloc(BUFSIZE2);
10   free(buf1R1); free(buf2R1);
11   buf1R2 = (char *) malloc(BUFSIZE1);   // Chunk allocated that likely overlaps
12   strncpy(buf1R2, argv[1], BUFSIZE1-1); // User input written into overlapping chunk
13   free(buf2R1); free(buf1R2);
14 }
```

Without fully understanding exactly how it could be exploited, the second call of `free(buf2R1)` on line 14 could lead to undefined behaviour. It's implied that when `argv[1]` is pushed onto the heap it can be used to overwrite a function pointer in heap metadata. In any case, once again the equivalent Rust does not compile, since once again this represents a `use after move`. This time on line 9, with the `buf2R1` value being moved out of scope on line 7. One last CVE to hammer home the point, consider CVE-2023-25136[10], a pre-auth attack against OpenSSH leading to DoS and potential RCE[6]. If only OpenSSH were written in Rust.

```rust
fn main() {
    let args: Vec<String> = std::env::args().collect();
    const BUFSIZE1: usize = 512;
    const BUFSIZE2: usize = (BUFSIZE1 / 2) - 8;
    let buf1R1 = Box::new([0u8; BUFSIZE2]);
    let buf2R1 = Box::new([0u8; BUFSIZE2]);
    drop(buf1R1); drop(buf2R1);
    let buf1R2 = Box::new(args[1].clone());
    drop(buf2R1); drop(buf1R2); // <-- value used here after move
}
```

2) A state machine model can be used for detecting possibly-interfering memory accesses dynamically, such as in the original Eraser design. States are Virgin, Exclusive, Shared and Shared Modified, with transitions for read, write and thread creation. Considering a bank account example where more than one thread may adjust a bank balance (`balance += adjustment`), explain how the Rust language features mentioned above may help to improve the security of code. What additional features may be needed?

There are strong similarities between this problem and the code sample provided on page 4. Making the appropriate alteration where we have a large number of threads making adjustments we get the following:

```c
#include <pthread.h>
#include <stdio.h>

void* adjust(void *args) {
    int **int_args = (int**)args;
    int *balance = int_args[0]; int *adjustment = int_args[1];
    *balance += *adjustment;
}
void main() {
    int balance = 20000; int adjustment = -1; int *args[2] = {&balance, &adjustment};
    pthread_t threads[20000];
    for (int i = 0; i < 20000; i++) {
        pthread_create(&threads[i], NULL, adjust, args);
    }
    for (int i = 0; i < 20000; i++) {
        pthread_join(threads[i], NULL);
    }
    printf("Balance: %d\n", balance);
}
```

We imagine an account with a balance of 20,000 that makes 20,000 payments that each take 1 from the balance. The overhead of the creation of a new thread every time makes the race condition much rarer, but the program still prints a value non-deterministically slightly more than 0 every time, giving the account holder an extra-money glitch. The explanation provided on page 5 for why this race condition occurs is fundamentally unchanged. It is comforting knowing that the equivalent Rust will not compile, as we learned earlier, thanks to Rusts move semantic. However it would be nice if we could get some version of this program to work. The missing additional feature as alluded to in the question are primitives to enforce thread safety by providing a locking mechanism. The Eraser design provides an interesting detective control against data races, however per the doctrine of Correctness-By-Construction it would surely be better to write programs where the `Shared-Modified` state was unreachable. This can be achieved in both Rust and C using thread safety primitives, in Rust specifically this would be Atomic Reference Counting (`Arc`) and `Mutex`.

3) Give an example of a different security or security-related feature (it could be one discussed in the Secure Programming course) that you can imagine might be usefully built on top of ownership tracking. Sketch how this might work.

Ownership of code in large collaborative repositories is a major issue. Large open source projects may benefit from a `CODEOWNERS` style model. This entails every directory in a project having an `OWNERS` file. Individuals or groups named in the file shall have the ability to edit any file in the directory cascading down, including the `OWNERS` file itself. A maintainer with their name in the `OWNERS` file in the root directory of the project can retain ultimate control of the repository by adding and revoking privileges in `OWNERS` files lower down the hierarchy, but for the most part can delegate the task of editing and approving alterations to the codebase to trusted editors lower down the hierarchy. Therefore making the approach more horizontally scalable. The only viable alternatives for a large project is making many users into top level maintainers, or having a hierarchy of users aggregate smaller changes into larger single changes, to make them more digestible by top level maintainers, as is the case with the Linux kernel. Both of these raise concerns re. the extent to which higher level maintainers actually understand what they are approving.

This becomes relevant to security when we consider supply-chain compromise. The theory goes that if ownership and responsibility is delegated to a wider pool of people, who have been selected according to a hierarchical chain of trust, each of whom has responsibility for a much smaller segment of a project, they are less likely to fall victim to fatigue in terms of pull requests, or approve something they don't completely understand as a result of being unable to keep pace with the expanse of their codebase.

This is neither a new nor original idea. A system that works a great deal like this is usually in use in corporate environments, and GitHub has had it as part of its product since at least 2017[13]. However the extent of adoption varies. It is notably not in use for the Rust language, nor the Linux kernel. Perhaps it is not the right approach universally, but particularly in light of the `xzutils`[3] compromise, it is surely worth the conversation whether more repositories would benefit from this style of ownership model.

4) Although Rust is type-safe, it has an unsafe keyword which provides an escape hatch for writing code that can manipulate pointers and values with fewer restrictions. The Rust book calls these "unsafe superpowers". Investigate this further and discuss the implications of including the unsafe feature.

Rationale:

- Rust provides a Foreign Function Interface (FFI)[17] to allow it to use functions from other languages that have an Application Binary Interface (ABI), notably C. This would have been useful in the early development of Rust, when not many crates were yet available. This also continues to prove useful for developers looking to integrate Rust incrementally into existing codebases that use a language with an ABI. Notably the Linux kernel[5]. Becuase the behaviour of an external function cannot be verified by the Rust compiler, it must be regarded as unsafe. Therefore, without the `unsafe` feature, the FFI could not have been included in a logically consistent way.

- Rust is self-hosted, ie. the Rust compiler is written in Rust. Within the Rust compiler you will find extensive use of the `unsafe` keyword. This is to facilitate lower level memory management and optimisations to give Rust C-like performance. Without this it would be difficult to take it seriously as a systems-level language, which is what it aspires to be. A good example can be found in the `vec` type, which implements a `truncate` method[14]. This reduces the size of the `vec` to that specified in a `len` argument. It is unsafe because when it does so it invokes `ptr::drop_in_place()`, itself an unsafe function that directly manipulates memory, and produces undefined behaviour if what it is trying to drop is null or invalid for read or write.

Implications:

In order to fulfill its promise of being a low latency systems language, and to replace languages that are inherently memory unsafe in an incremental fashion, Rust compromises on its own promises of memory safety. According to The Rust Foundation 20% of crates make direct use of the unsafe keyword, and over a third make direct function calls to another crate that uses the `unsafe` keyword[18]. The prevalence of so much unsafe Rust undermines the image of the language as offering a memory safe platform. On balance however it feels unwise to let the perfect be the enemy of the good. The overwhelming majority of all Rust ever written is safe. Without going out of their way to do so a developer cannot write *new* unsafe Rust, and security researchers seeking to verify existing Rust and find vulnerabilities know exactly where to look. Further, the language developers have adopted strong conventions for writing unsafe functions, with comments above functions clearly explaining how they should be invoked to not cause undefined behaviour.

Part C: Designing a Secure System

1) Using a diagram to help explain, give a high level view of your system design including the various software components involved and messaging between them. Software may run on multiple devices, at a minimum you should consider the back end cloud-based servers, front end web interface, and connection to fitness tracking apps running on mobile devices (focus on particular platforms in each case to consider specifics). Describe and justify programming language choices for each component and explain communication paths.

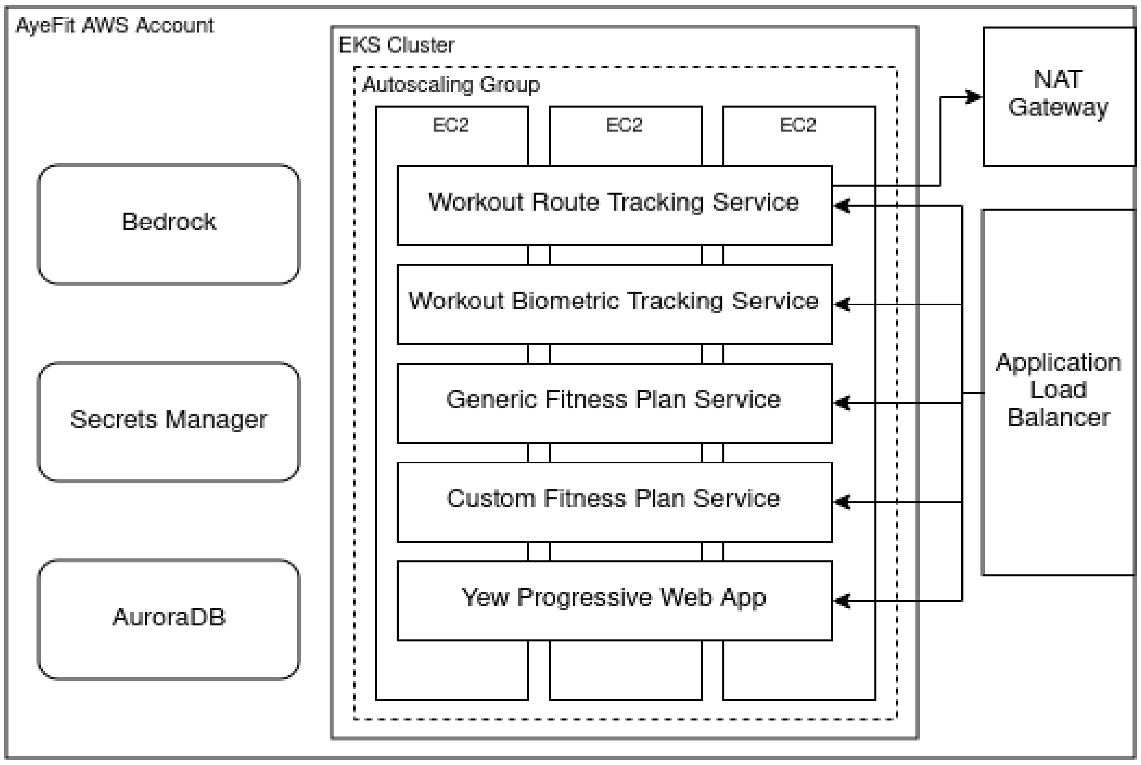
AyeFit AWS Account
EKS Cluster
Autoscaling Group
EC2
EC2
EC2
Bedrock
Secrets Manager
AuroraDB
Workout Route Tracking Service
Workout Biometric Tracking Service
Generic Fitness Plan Service
Custom Fitness Plan Service
Yew Progressive Web App
NAT Gateway
Application Load Balancer

Figure 1: *Cloud-based back end for AyeFit.*

There are several simplifications and omissions in Figure: 1.

- The EC2s are all in a private subnet providing network layer segregation from beyond the AWS account.
- Lines to the 3 AWS services on the left are omitted, since obviously only the Custom Fitness Plan Service will communicate with Bedrock (AWS Foundation model service) and every app will make use of Secrets Manager and AuroraDB.
- Yew is a web app framework for Rust, that can compile to WASM and be run in the browser. WASM based frameworks have come of age in the last few years, and the *AyeFit* team have decided the Rust-WASM Yew framework is suitable for their needs.
- The decision has been made to run the web app as a Progressive Web App (PWA). This means the same app will be run in the browser and deployed to the iOS and Android app stores. This is to

minimise the total amount of code in use at *AyeFit*, and therefore reduce the attack surface for code based vulnerabilities.

- Each of the services, including the Yew PWA, are Kubernetes services, and are comprised of a Kubernetes Deployment, Ingress, and Horizontal Pod Autoscaler (HPA).
- The EC2s in the Autoscaling Group (ASG) are balanced across multiple availability zones (AZ) within the AWS region to provide high availability in the event of AZ failure.
- The Application Load Balancer (ALB) is public facing and routable from the internet, providing a means for the APIs exposed by the many services to be reached by devices.
- The PWA service is greatly simplified, with it actually being comprised of several microservices for handling various functions including password reset, authentication and token renewal.
- Considerable additional plumbing is present in the cluster and account that would be too difficult to compress into one diagram. Additional items include but are not limited to:
  - Athena for centralised log aggregation
  - Managed Prometheus for observability
  - Cert-manager for cluster certificate management
  - Cloudtrail for API action audit
  - GuardDuty for security detection and posture management
  - GitHub providing source control, CI pipelines and Infrastructure as Code

The front end counterpart to this architecture is uncomplicated at a high level. A few key features are worth a mention:

- A service worker component exists to govern the apps functionality when offline for any amount of time, as is required to make this app an installable PWA.
- A simple key-value store is in use to act as a local DB while offline or buffering data during a workout.
- Logic exists to make API calls to a maps API and to the health app on the device. The health app API calls simply require on-device permissions to be granted, the API calls to the mapping service require a session key which the front end requests from the Workout Route Tracking Service in the backend, which in turn retrieves the session token by making a call itself to the mapping API, leveraging a more permanent long lived token stored in the Secrets Manager. This run to the mapping API is what necessitates a NAT gateway in the account, so that the Workout Tracking Service can reach IPs beyond its subnet.
- A very standard authentication flow is observed when the client logs in and authenticates their session. Having established a TLS session and authenticated via username and password the user is issued with a JSON Web Token (JWT) that can be refreshed and used as bearer authentication for the other APIs.

Rust has been chosen as the language to comprise the full stack of this application, and all its related services. This has been done for three reasons:

1. The founders of *AyeFit* wanted to write an app that would be extremely unlikely to ever compromise user data. They abide by the philosophy that the earlier an investment is made towards this endeavour, the cheaper it will be to uphold this aspiration in the long run. Rust was perceived by them as being a good choice for providing safety from security issues relating to overflow attacks and data races.

2. The founders knew that the architecture would need to scale horizontally with user traffic. They also knew this would be very expensive in the cloud, and therefore they looked at an early stage to run the most efficient and performant application possible, in the interest of minimising running cost.

3. Rusts approach to concurrency convinced them that it much less likely to have embarrassing runtime race conditions and cause outages as a result.

The only exception to this is the use of Terraform for the provisioning of the infrastructure in GitOps. Rust did not seem the appropriate choice here since it is not first and foremost a declarative ”desired-state” language, as Terraform is.
2) Define a minimum of four core use cases based on the scenario above. Emphasise any security features involved in providing those uses and how they are implemented.

- **Interactive Fitness Plan Builder:** Based on an interaction with an LLM the app will build both short and longer term fitness plans for a user, determined by what the users stated fitness goals are, and moderated by limitations the LLM will know the user to have (eg. weight, age, susceptibility to asthma). There are significant security and privacy concerns with this feature. The confidentiality and integrity of user data is considered to be of the greatest importance. For quality control, and so the user can reference the building of their own plan, the conversation will be stored. It will be anonymised to the greatest extent possible, with the primary key of the table it is stored in being an ID, with personally identifiable information (PII) removed and placed in a separate table that *AyeFit* employees will never directly access. The data shared by the user with *AyeFit* will be encrypted in transit via TLS, at rest once by the cloud service provider to mitigate hardware theft, and then at rest once again to mitigate cloud account compromise. Each row of the table containing user PII shall be encrypted with a key derived from the users account password. This provides a zero-trust architecture in the style of a password manager, meaning even in the event of a catastrophic data leak, it will still be impossible to deanonymise users, provided they had strong passwords.

- **Geolocation Workout Tracker:** Using location data gathered from a users device in real time *AyeFit* will provide post workout summaries of what the user did in relation to where they exercised. This being relevant to sessions where a user went for an outdoor run or cycle rather than remaining in a gym. This carries many of the same considerations as the feature for custom training programs. The data collected and stored is potentially of a very sensitive nature, since it potentially reveals where the user lives and could even disclose their real time location. Because of safety concerns around this, the app will not broadcast location data to the backend continuously. A full dump of the outing will only be made once the workout is over. Again, the data involved shall be encrypted in transit and at rest.

- **Fitness Plan Catalogue:** This feature shall provide users with access to non-personalised generic fitness plans as an alternative to those created with the personalised planner. The plans shall be licensed and should not be used by competitors, however given that they will be available to all users of the platform in the same form they can hardly be regarded as confidential. Their integrity is considered important, since overly ambitious plans could be dangerous if users trust them implicitly and overexert themselves as a result.

- **Biometric tracking:** As an extension of the app, wearable devices shall be leveraged to collect biometric data during exercise sessions. *AyeFit* itself will not collect this data, and will instead request it from existing device native fitness trackers, eg. the Apple Health App. This will entail persistent cross-app privileges being granted to *AyeFit* which will effectively make it a custodian of the biometric data. In terms of sensitivity this will be treated exactly the same as the information the user provides to the plan builder, since the data is similarly sensitive.

3) Produce a simple threat analysis of the system, considering at least five different threats to security or privacy. You may use STRIDE or a similar approach, but start from potential damage that could be caused by a malicious actor and consider possible motives and opportunities, rather than starting from potential vulnerabilities.

The following ideas are inspired by STRIDE, and a trust boundary oriented approach to threat enumeration:

- **Information Theft (due to cloud account compromise):** A financially motivated threat actor could act on the trust boundary either between the GitOps server and the AWS account, or the internet and the Application Load Balancer, with the intention of exploiting a vulnerability to gain unauthorised access to the AWS Cloud Account and deploy ransomware.
- **Denial of Service:** A lone chaotic actor looking to prove/test their abilities could again attempt to exploit the trust boundary between the internet and the ALB, this time by mounting a denial of service attack.
- **Malicious Plan Alterations:** A malicious employee within the *AyeFit* organisation, motivated by what they perceive to be unfair compensation for their work, could exploit the trust boundary between the GitOps server and the cloud account once more. They could introduce alterations to the Fitness Plans that could be harmful to *AyeFit's* reputation.
- **User Deanonymisation:** An admin could leverage special privileges they have to log in as the root user via the console to obtain the keys in the secrets manager required to deanonymise users. Motivated by a desire to debug an insidious and longstanding issue.
- **User Account Takeover:** A foreign entity looking to cause mischief in the run up to a political event leverages credentials that have leaked in recent major data breaches, and have tried them for as many users as they can in the hope that users reuse passwords. Once they gain unauthorised access they deface the user accounts causing reputational damage to *AyeFit*.
- **LLM Related Security Concerns:** A malicious engineer hellbent on being a nuisance has compromised the pre-prompt of the LLM being leveraged in the AWS Bedrock service, causing it to tell users that they are perfect the way they are and don't need a fitness app to feel good about themselves. This exploits the trust boundary between the GitOps change management process and the AWS account.

4) From your threat analysis, suggest three core abuse cases you would want to prevent, describing how a countermeasure or system response will prevent an attack succeeding.

- **Cloud account compromise resulting in ransomware:** A number of strong mitigations exist for this, all of which should be deployed:
  - A detection and response operation should be in place leveraging detective controls from GuardDuty and SecurityHub.
  - Static and Dynamic analysis of code should occur as part of the build and release process from GitOps, to mitigate the risk of vulnerabilities in the internet facing API endpoints.
  - Scanning of container images at both build and runtime for known vulnerabilities, eg. Alpine Linux images that provide the base for the services.
  - Regular scheduled penetration tests against the cloud environment should be conducted by external third parties to validate that the environment is not (easily) exploitable by some unforeseen means.

  - A regular schedule should be established for backups to be committed to S3 buckets in a WORM (write once read many) compliant fashion using object locking.

- **User Deanonymisation:** A serious concern, for which only one major mitigation has been deployed. Namely applying a zero trust approach to the encryption of the user PII table.
- **DoS:** Probably the thing which the organisation regards as most likely to cause harm to *AyeFit*, DoS will largely be mitigated by the protections provided by AWS Shield as part of the standard feature set of an Application Load Balancer. This product leverages anomaly detection to dynamically filter packets that are considered to potentially be part of a DoS attack.

5) Initially the AyeFit platform was seen as providing individual training plans but the company rapidly realised that social features would increase subscription uptake. AyeFit now want to add ways to compare and contrast fitness plans and progress with those with other training friends on AyeFit. What additional security threats and requirements might arise in this situation and how should AyeFit address them?

A likely feature request that will come of this new product offering will be the ability to share and publish the Geolocation data associated with workouts. This may well be the users intention and desire, but we will still make it not the case by default, since this will greatly infringe on the users privacy and should not be shared lightly or by accident. Ultimately no obvious further security measures could or should be taken, if a user wishes to publish there precise location data it is not *AyeFit's* place to tell them to do otherwise. A precisely analogous argument applies for sharing of workout biometric data. Again, this shall be disabled as a strong default, but the user may be free to share it if they wish.

Social features will likely entail direct messaging, following and posting. All of this inevitably entails peer-to-peer abuse on the platform, with a significant risk of cyber stalking to users. An acceptable use policy should be drafted, and machine learning, account review and reporting processes should be introduced to enforce it, to provide for user safety on the platform.

By the same mechanism *AyeFit* will become a potential vector for malicious URLs and phishing links. They should be mitigated by the same means. Potentially with external URLs subject to a strict whitelist, and repeat offenders for disseminating URLs being cautioned and eventually banned.

Social features will entail a significant rise in user input, and will provide multiple portals for user input. With each of these great care must be taken to sanitise this input to mitigate the risk of injection attacks, notably SQLi. This should be achieved through the use of prepared statements, and a special focus on the separation of concerns in code review, ie. the distinction between data and logic.

The creation of a social platform has the potential to influence a large group of people, which will potentially make it attractive to groups who wish to subvert democratic processes, by insidiously harvesting data, or even directly influencing people through targeted posts. Tracking across other social media platforms should be prevented to the best of our ability, and reCaptcha's & machine learning should be leveraged to try and stem the proliferation of bots.

In addition to the more novel threats mentioned above, the developers of *AyeFit* will also need to cope with the more mundane reality of user numbers increasing, the cost of running their security tooling increasing, their attack surface in terms of code volume increasing, and the activity per user increasing. All of this will make the scope and scale of the *AyeFit* security function more expansive.

References

[1] Go grpc quickstart. https://grpc.io/docs/languages/go/quickstart/.

[2] T. Bordis, T. Runge, A. Kittelmann, and I. Schaefer. Correctness-by-construction: An overview of the corc ecosystem. *Ada Lett.*, 42(2):75–78, apr 2023.

[3] U. CISA. Cisa xzutils, 2024. https://www.cisa.gov/news-events/alerts/2024/03/29/reported-supply-chain-compromise-affecting-xz-utils-data-compression-library-cve-2024-3094.

[4] T. Coquand. Coq theorem prover. https://github.com/coq/coq?tab=readme-ov-file.

[5] R. for Linux. Rust for linux, 2024. https://rust-for-linux.com/.

[6] Y. Mizrahi. Openssh pre-auth double free, 2023. https://jfrog.com/blog/openssh-pre-auth-double-free-cve-2023-25136-writeup-and-proof-of-concept/.

[7] D. Norfolk. Mathematical approaches to managing defects, 2006. https://www.theregister.com/2006/08/14/math_managing_defects?page=3.

[8] NVD. Cve-2016-5195, 2016. https://nvd.nist.gov/vuln/detail/CVE-2016-5195.

[9] NVD. Cve-2022-0609, 2022. https://nvd.nist.gov/vuln/detail/CVE-2022-0609.

[10] NVD. Cve-2023-25136, 2023. https://nvd.nist.gov/vuln/detail/CVE-2023-25136.

[11] OWASP. Owasp samm v2.0.3, 2022. https://owaspsamm.org/model/.

[12] OWASP. Double freeing memory, 2024. https://owasp.org/www-community/vulnerabilities/Doubly_freeing_memory.

[13] J. Pace. Github codeowners, 2017. https://github.blog/2017-07-06-introducing-code-owners/.

[14] rustlang. rust. https://github.com/rust-lang/rust/blob/55cac26a9ef17da1c9c77c0816e88e178b7cc5dd/library/alloc/src/vec/mod.rs#L1219C44-L1219C45.

[15] Speykious. Vulerabilities in safe rust, 2024. https://github.com/Speykious/cve-rs/blob/main/src/use_after_free.rs.

[16] A. Taylor, B. Nowierski, K. Hara, and MiraclePtr. Use-after-freedom: Miracleptr, 2022. https://security.googleblog.com/2022/09/use-after-freedom-miracleptr.html.

[17] T. R. F. Team. Rust ffi - rust by example. https://doc.rust-lang.org/rust-by-example/std_misc/ffi.html.

[18] T. R. F. Team. Unsafe rust in the wild. https://foundation.rust-lang.org/news/unsafe-rust-in-the-wild-notes-on-the-current-state-of-unsafe-rust/.

[19] A. Weidemann. Google tag chrome rce update, 2022. https://blog.google/threat-analysis-group/countering-threats-north-korea/.