Response From Michael Justus

Context: https://www.my-course.co.uk/mod/hsuforum/discuss.php?d=270896

Hi Shan,

The use of a Structured Activity is intelligent and shows a deeper understanding of UML.

The activity "Copy Token from URL" implies the generation of a token value. I assume this is where the failure occurs, referred to in the post? If so, out of interest, what is the likelihood of six sequential token values occurring? The Java Documentation mentions that java.util.random is "not cryptographically secure", and the recommendation is to use instead "SecureRandom" (Java, 2021) because it adheres to FIPS 140-2 standard (section 4.9.1, "Power Up Tests", FIPS (2001))

You raised a rather intriguing recommendation in the post to utilise a microservice approach when generating RNG-based tokens. How would such an approach work if, for example, a library intends to generate a cryptographically secure token of, say, 128 bytes in length? I suppose even the microservices would require some form of limits to prevent overloading them with requests or the potential to produce sequential numbers from their internal RNGs.

References

FIPS (2001) FIPS PUB Security Requirements for Cryptographic Modules. Available from https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.140-2.pdf [Accessed 18 Aug 2021]

Java (2021) Java Platform Standard Edition 8: Class SecureRandom. Available from https://docs.oracle.com/javase/8/docs/api/java/security/SecureRandom.html [Accessed 18 Aug 2021].

Response To Michael Justus

Hi Michael,

Thanks for your response. The reason why it's possible to obtain six sequential values is because of how the java.util.Random library is built. If a developer wishes to use the library, they'll begin by creating an instance of Random itself, which will then set up an internal state. When a developer wants to get some random value, they would call one of the "next" methods (e.g. nextInt, nextFloat, nextBytes). Due to the internal state of the RNG, future results are always influenced by previous results, so the "next" methods effectively create a deterministic sequence. In the context of a password reset token, a developer might choose to use nextBytes() to generate a token, however, this would mean that if an attacker requests a password reset six times in a row, they're effectively calling nextBytes() six times in a row, providing a set of sequential values which can be used to determine the generator's internal state. This does make the assumption that no other users request a password reset within that timeframe, however, in certain cases, it's still possible to determine internal state even if values are skipped (tailcall, 2017).

With regards to your second point, I know of a quantum RNG that is offered via a web API, which has been implemented in different libraries and languages (ANU QRNG, n.d.). Although it's possible to do the same for a cryptographically secure RNG, it would introduce network-based risks that need to be managed. In my opinion, those risks outweigh the benefits, because it would be creating a (subtle) single point of failure- what do you think? Microservices would improve security when combined with a secure offline library so that each instance is truly standalone, but as you mention, additions would be necessary. Apart from ensuring that each microservice instance has a unique, truly random internal state, implementing a rate limiting strategy is a good suggestion as it would reduce the risk of a single user obtaining a sequence of values and has the additional benefit of improving overall resilience (Google, 2021). I imagine you're thinking something along the lines of allowing each unique IP to send one reset request every X hours? Another key idea is load balancing- if requests are consistently distributed across different microservice instances, and each instance has a truly random internal state, it wouldn't be possible for an attacker to observe sequential tokens.

References

ANU QRNG. (n.d.) Frequently asked questions. Available from: https://qrng.anu.edu.au/contact/faq/#downloads [Accessed 18 August 2021].

Google. (2021) Rate-limiting strategies and techniques. Available from: https://cloud.google.com/architecture/rate-limiting-strategies-techniques [Accessed 18 August 2021].

tailcall. (2017) Cracking RNGs: Linear Congruential Generators. Available from: http://tailcall.net/posts/cracking-rngs-lcgs/ [Accessed 18 August 2021].

Response From Michael Justus

Hi Shan,

I find the response excellent because you clearly show the underlying reason why such token generation can fail. The reference to the quantum RNG project is also a valued addition because (reading about the project) is a fascinating product developed by Down Unddah, whose goal is to produce almost random numbers based on some quantum field fluctuations.

Regarding the use of microservices in generating RNG-based tokens, with the ANU project referenced, it is very likely these two components can quickly resolve the issue identified. Each microservice contains a unique state, and the yonder machine generates such uniqueness. But, networks introduce points of failure. After all, attacks inevitably involve network access (yes, if attackers are internal to an organisation, with direct access to hardware and applications, that is another concern). So one thought is that leveraging microservices must be hardened against failure (security or otherwise). Rudrabhatla (2020) makes a case for three pertinent microservice concerns: secure the attack surface, encrypting everything; and, continuous monitoring. Jindal et al. (2019) put forward approaches such as sandboxing, determining microservice capacity and using load generators like Kubernetes Cluster.

References

Jindal, A., Podolskiy, V. & Gerndt, M. (2019) Performance modelling for cloud microservice applications. Proceedings of the 2019 acm/spec international conference on performance engineering: 25-32.

Rudrabhatla, C.K. (2020) Security Design Patterns in Distributed Microservice Architecture. arXiv:2008.03395.

Response From Andrey Smirnov

Hi Michael,

Thank you for providing those references, I found them extremely interesting to go through. Indeed, with the adoption of distributed architectures we have seen a widening of the attack spectrum, and many security techniques that were successful with monolithic web applications are not as effective when applied to microservices. One approach towards increasing MSA security that has seen good adoption in the industry revolves around the use of token-based authentication/authorization mechanisms (e.g., OAuth, JSON Web Tokens), particularly coupled with the API gateway pattern. In this design approach, the API gateway acts as a single point of entry into the system and enforces token validation before routing the frontend requests to backend services. Some organizations that deal with sensitive customer data (ABN AMRO being one example) take this concept one step further and employ a secondary (Internal/Private) API gateway that adds another layer of security. This can mean that every request from the frontend application needs to pass through up to 4 layers before ending up in the actual backend service responsible for handling the call. While this setup arguably adds latency costs, the benefits from the "defence in depth" approach, for some organizations, largely outweigh these costs.

Kind regards,
Andrey

Response From Cathryn Peoples

Excellent introduction into this discussion, Shan.

Excellent evidence of your knowledge and understanding here.

The process leading to the weakness occurring clearly communicated through the UML model. Well done, Shan.