Integrated InfoSec submission to the RFC on the draft Secure Software Development Attestation Common Form Explanatory Annex 12/2023

[1] Examples of hazards that the scope of the attestation would not assuredly cover

Heartbleed (2012 to present)
https://theconversation.com/how-the-heartbleed-bug-reveals-a-flaw-in-online-security-25536

At the end of December 2011 Robin Seggelmann submitted to OpenSSL an extension to the TLS and DTLS protocols providing a "keep alive" mechanism to maintain a connection without requiring repeated re-negotiation.

In February 2012 Seggelmann's contribution was adopted as RFC 6520, authored by himself with (seemingly) two co-authors and apparently reviewed by nine others. The extension was incorporated into the June 2012 release of OpenSSL. The extension included a critical vulnerability which remained unreported until April 2014. This could be used easily to exfiltrate potentially sensitive information such as encryption keys from both servers and security appliances implementing OpenSSL, and it is likely to have been exploited widely prior to its being reported. The full extent of its abuse is still not fully known and many systems probably remain unpatched and vulnerable to this day.

From April 2014 public attention was focused on the bug in OpenSSL (ostensibly a coding error) but the real root cause was a conceptual error in the design of the extension itself. It used a challenge/response across a connection, in which the challenger provided a message of arbitrary length and a separately specified indicator of the length of the message, both of which the responder echoed back. It was entirely possible for length indicator to misrepresent the real length, and if it was set to a notional length greater than the actual length of the message, the responder echoed back the message plus the content of memory following it up to the misrepresented length. This was a fragile, abusable mechanism, which was nevertheless ratified by multiple reviewers as sufficiently robust on its path to adoption as an RFC.

Quite apart from the specific weakness described above the RFC document has several weaknesses. It specifies a seemingly excessive maximum message length of 16kB together with the ambiguous statement "*If the payload_length of a received HeartbeatMessage is too large, the received HeartbeatMessage MUST be discarded silently*", suggesting that messages longer than this maximum should be discarded but making no reference to the length specification parameter. Ironically, it also states "*If a received HeartbeatResponse message does not contain the expected payload, the message MUST be discarded silently.*" In the adversarial context, this is of course advice to the malicious actor, who will probably ignore it. Nevertheless, the authors of the RFC considered that "*[t]his document does not introduce any new security considerations*" – a clear indication of the common erroneous position that "security" is an add-on or afterthought.

Supposing the extension definition had specified a fixed length message there would have been no requirement for the hazardous separately specified length parameter, and the vulnerability could not have existed. But. crucially, the unnecessary complexity and resulting fragility of a hazardous mechanism passed scrutiny at several stages: at the point of submission to the OpenSSL project, at the point of being accepted as an RFC, and at the point where it was included in the release of OpenSSL. Clearly, the review process at every stage, though present, was insufficient to detect the hazard.

Although the Heartbleed bug in OpenSSL attracted much publicity and has been addressed by patching, the ambiguity of RFC 6520 and the inherent fragility of the mechanism it defined remain a hazard to any other implementation of SSL that follows the RFC, as it remains current as a 'proposed standard'.

It is hard to see how the attestation as drafted would address an issue such as this, as it is grounded in a conceptual error that persisted from initial design through the entire development and implementation cycle despite review at multiple points and the root cause appears to remain unchallenged to date despite a highly publicised adverse outcome.

Collapse of Microsoft Azure (29 Feb 2012)
https://azure.microsoft.com/en-us/blog/summary-of-windows-azure-service-disruption-on-feb-29th-2012/

The Azure cloud service relies on an extensive system of digital certificates that authenticate data flows around the world-wide network. On the leap day 29 February 2012, it suffered a multi-continent collapse of service for around 36 hours because these certificates (having a one year life) were recreated en mass with a renewal date of 29 February 2013. This date of course did not exist, rendering all the certificates invalid. The collapse was directly triggered by some legitimate administration that caused the certificates to be regenerated, but the issue that drove the collapse had lain dormant for an indeterminate time prior to this.

It turned out that some developer had specified that the renewal date of certificates should be calculated by merely adding one to the year of the creation date, despite there being established and proven date calculation methods available. This poor decision remained undetected and dormant until, by chance, a mass update of certificates occurred on a leap day.

The key point here is that the error was not detected by testing, nor did it show up during operational generation of certificates until that critical date. The underlying problem was poor engineering judgement at coding time that failed to take account of a readily predictable edge case.

It is hard to see how the attestation as drafted would address the root causes of vulnerabilities such as this, as it presumably survived code review and testing by an experienced vendor, and also deployment and operations without being detected until the critical date.

UK Air Traffic Control Flight Plan Reception Suite Automated Sub-system Incident (28 Aug 2023)
https://publicapps.caa.co.uk/docs/33/NERL%20Major%20Incident%20Investigation%20Preliminary%20Report.pdf

On 28 August 2023 the UK Air Traffic Control (ATC) automated flight plan processing system shut down for around seven hours, delaying 575 flights and incurring an aggregate of over 1000 hours delay to flights. The cause was an inappropriate response to an anomaly in a submitted flight plan.

It is well known that there are navigational waypoints with duplicate names and these are recognised as a potential contributory factor to accidents (e.g. American Airlines Flight 965, December 20, 1995). So when the system encountered a flight plan including two identically named waypoints it quite properly took defensive action. However, that action was to throw a critical error and shut down, passing operation to its backup system. The backup system took over processing, encountered the same anomaly in the same flight plan, threw a critical error and also shut down. Manual intervention was required to recover functionality, resulting in the losses described above.

Duplication of waypoint names is being eliminated, but duplicates still exist. So the system was, reasonably, designed to recognise this in order to avoid hazardous flight plans being approved. The error was in the nature of the response, choice of which demonstrates a conceptual disconnect from the real priorities facing ATC. The incident report states: "*Clearly a better way to handle this specific logic error would be for FPRSA-R to identify and remove the message and avoid a critical exception* [leading to shutdown]*. However, since flight data is safety critical information that is passed to ATCOs the system must be sure it is correct and could not do so in this case. It therefore stopped operating, avoiding any opportunity for incorrect data being passed to a controller. The change to the software will now remove the need for a critical exception to be raised in these specific circumstances.*" A bad judgement of this nature would be unlikely to be discovered by the software testing regimes covered by the attestation or its supporting standards. It is basically failure to properly recognise during the design phase the priorities of the operational context.

However the story is potentially not over as, under "*Actions taken or underway to prevent the interruption recurring*" the incident report states inter alia "*A permanent software change by the manufacturer within the FPRSA-R sub-system which will prevent the critical exception from recurring for any flight plan that triggers the conditions that led to the incident. This change will*

prevent the software from finding a duplicate waypoint that could cause an incident." [my emphasis]. I am not an ATC systems developer, but from my experience as a general systems engineer this seems dangerously ambiguous, as it could be construed as meaning potentially hazardous duplicate waypoints would be ignored.

The key issue (conceptual failure at the design stage) seems to lie outside the scope of the attestation.

[2] Standards of performance

In respect of practicably attainable standards I quote from the *Integrated InfoSec submission to the 2023 UK DCMS call for views on software resilience and security for businesses and organisations*.

"An informative example of the problem is revealed by examination of reports by the UK Huawei Cyber Security Evaluation Centre Oversight Board. The Board identified *'[e]xtensive non-adherence to basic secure coding practices, including Huawei's own internal standard[...]'*, manifested by high prevalence of well known and very basic machine level coding errors that can result in attackable vulnerabilities, and uncontrolled use of third party components including inter alia *'70 full copies of 4 different OpenSSL versions, ranging from 0.9.8 to 1.0.2k (including one from a vendor SDK) with partial copies of 14 versions, ranging from 0.9.7d to 1.0.2k[...].'* (HCSEC 2019), i.e. simultaneous use of fragments of code from multiple releases of the same software. This is a clear indicator of completely unmanaged development processes.

Disturbingly but unfortunately predictably, the 2021 report found that there was *'no overall improvement over the course of 2020 to meet the product software engineering and cyber security quality expected by the NCSC'*, but *'[s]ustained progress has been made during 2020 on remediating the point-issues found in previous reports.'* (HCSEC 2021) This typifies the common approach to improvement – post facto point fixes of specific issues that result from poor practice, without addressing the underlying root causes of the poor practice. Inevitably, this approach disallows real improvement. Huawei is cited here as an example only, because independent evaluation has been undertaken. It is not safe to assume that higher standards would necessarily be found were any other major developer/vendor comparably audited."

The obvious key message here is that the nature of the processes designed to prevent, detect and correct errors and of course adherence to them, not just the presence of those processes, is vital to achieving the required results.

[3] Influence of methodologies on security

In respect of the influence of development methodologies on security I quote from the same submission.

"Development practices such as agile, widely adopted for code production efficiency, have tended to compress the development lifecycle from both ends, progressing directly from concept ('user stories') to implementation (coding). This de-emphasises the design stage where considerations of robustness and fitness of algorithms to their intended purposes traditionally take place (as does, for example, consideration of properties such as strength of materials in structural engineering). It also obscures entire attributes of a project that can not be explicitly expressed in 'user stories' (most significantly, security), which causes them not to be considered at all when the code is written, or to be implemented in a chaotic piecemeal fashion.

At the very worst, such methods as 'pair programming', where two people code side by side on one workstation discussing implementation as the code is typed in, are entirely devoid of any formal consideration of design. Indeed one of the supposed advantages of the technique is described by a somewhat laudatory page on Wikipedia as *'[t]he benefit of pairing is greatest on tasks that the programmers do not fully understand before they begin"* i.e. where they launch into the primarily manipulative coding stage of the development process without having considered what they are actually trying to achieve. The inevitable consequence of what could more legitimately be called 'suck it and see coding' is unverified (and by virtue of the current scale of applications, increasingly unverifiable) code quality. Such practices would not be tolerated in any established branch of engineering.

The problem is exacerbated by inadequate developer understanding of first principles. Any brief perusal

of questions and responses on developer fora such as Stack Overflow clearly demonstrates a general lack of basic conceptual understanding even in quite simple cases where code security and robustness are not at issue. This respondent has not infrequently been told by collaborating developers that some required mechanism ‘can’t be done’ in some language merely because there is no ready made library method for it, whereas some independent pure language level coding has generally been sufficient to achieve it . Similarly, it is well recognised that many developers copy and paste code fragments from third parties with very little if any attention to how the code so used actually works. Far from there being any barrier within the open source community in this respect, it is clear that many deployers of open source components do not bother to examine the provided source code. They merely refer to the API, trusting the code itself just as closed source library methods are used as trusted building blocks. As a result, errors in library methods probably contribute at least as significantly to application vulnerabilities as errors explicitly made by application developers themselves. In the case of closed source libraries such vulnerabilities are likely to persist and spread due to licensing restrictions on reverse engineering the libraries."

Clearly, reliance on technical means (e.g. "secure methods", memory-safe languages and automated testing) to ensure security is insufficient unless supported by attention to design, development and testing by informed competent persons who accept responsibility for outcomes.

[4] Developer education

In respect of developer education in general I quote again from the same submission.

"[...] poor practice almost certainly derives at source from inappropriate approaches to developer training. Commercial developer training concentrates on the mere knobs and levers of specific languages, libraries and proprietary development systems, resulting in the ability to produce code but with no assurance of quality or, by virtue of the plethora of available high level methods and libraries, any understanding of what the executable code actually does internally, which is of course where exploitable vulnerabilities frequently reside.

Currently, first degree courses in software development fare little better. In 2016 this respondent researched this [...] It was found that a representative sample of UK software engineering degrees included secure coding, if at all, only in their final year (once insecure habits would inevitably already be ingrained), and typically as an optional module. A review for the present submission established that the position has hardly changed, except that a new generation of ‘cyber security’ degrees has emerged that profess to touch on secure coding. However it is unlikely that a prospective software developer will take a ‘cyber security’ degree as opposed to one explicitly in software development, so, regardless of any adequacy of the actual training provided by such new degrees, the appropriate community is not being addressed.

This is fundamentally a cultural problem. The current semantic emphasis on ‘coding’ as opposed to ‘programming’ is exemplary of an inappropriate mindset on the part of both educators and practitioners as it focuses on the manipulative rather than the conceptual aspects of the discipline. It is important to make a clear distinction between ‘secure coding’ and ‘defensive programming’. Several active developers this respondent has queried see the two as synonymous. In reality however, while secure coding primarily addresses technical errors or omissions when generating the source code of programs, defensive programming in addition includes attention to the robustness and appropriateness of algorithms and functionality at the design stage as well. As the formal design stage has become largely deprecated in current practice, failure to appreciate this is understandable but it is major contribution to vulnerabilities and thus insecurity.

[...]

To achieve adequate standards, the emphasis of developer training must be revised. It is entirely unrealistic to expect secure development practice from persons not trained at all in the necessary disciplines or exposed to them merely as an optional afterthought as at present. This is all the more significant when retraining those with substantial experience already in insecure practice as habits die hard. The necessary concepts must therefore be embedded seamlessly in the educative process from day one for new echelons of trainee developers, who will progressively succeed the currently inadequately trained.

All developer training on every topic at every stage must seamlessly incorporate discussion of the hazards and preventative strategies required to render the relevant code robust and secure as a matter of course. Training in 'insecure coding' should not lead to formal qualifications. There must also be a strong emphasis on the overarching engineering mindset – the conscious application of forethought, a logical analytical approach to problem solving based on first principles, and commitment to and responsibility for the robustness and safety of the delivered product. These attributes underpin professional practice in all established branches of engineering but are currently appreciably absent from the thinking of the general developer community.

[...]

In order to bring software vulnerability under control we need echelons of developers who by second nature design and program securely. Any alternative leaves us reactively applying successive layers of first aid while the patient expires. The content of curricula and methods of delivery both need to be revised to ensure that the teaching of 'insecure coding' is a thing of the past. No post-development interventions will improve the security of software in general. Unless software development is raised to the level of a genuine engineering discipline, the current state of insecurity will remain with us permanently. Government has a potential role in specifying and guiding the introduction of curricula and examinations that fulfil the necessary requirements."

As evidence of causal factors of poor developer education, I quote from the 2016 *Integrated InfoSec submission to the US Commission on Enhancing National Cybersecurity* (Obama EO 13718).
https://www.nist.gov/document/integratedinfosecrfiresponsepdf

"In August 2016 the US Department of Education published a fact sheet on a potentially very worthwhile initiative to grant access to education to low income students. However the offerings in software development it describes, two out of three of which are of 12 and 13 weeks duration respectively, cause us significant concern. Our greatest single objection is that the 13 week course in web development describes its outcome as preparing students "for jobs as mid-level software engineers." This is a completely unrealistic expectation if the word 'engineer' is interpreted in its normal professional sense. It takes from three to seven years of intensive post-high school education to train an entrant to any of the established engineering professions and around five years post-education practice to reach mid-level rank.

We consider that a short course of 13 weeks is only capable of familiarising students with the most basic generic programming structures (e.g. loops and branches), the 'knobs and levers' of a vendor-specific development system and the use, but not the mechanisms, of a limited subset of libraries. It cannot teach how to code except mechanically, and cannot possibly impart even the most elementary skills in defensive programming. Nevertheless, in the community at present, knowing the IDE and libraries equals 'developer', so the fault most likely lies primarily not with the definers of this course but with the culture within they have been obliged to define it.

The adverse outcome of such offerings is self evident. The top four vulnerabilities in the OWASP Top Ten prevalence ranking of web application vulnerabilities, have shuffled places from time to time but have communally remained at the top of the list for over a decade despite being widely discussed, and supposedly widely understood, by the developer community.

We also take issue with the drives of several national administrations to introduce 'coding' into the mandatory educational curriculum from early years. To redress a shortage of expertise for delivering this, there have been moves, at least in the UK, to provide crash courses in coding for existing teachers so they can deliver the curriculum without prior experience in the subject. The likely outcome is merely to release further echelons of inadequate software developers into an already broken profession.

[...]

We would add that, at least in the UK, the bachelors' degree does not at present typically pass muster as validation of skills in security. We examined the syllabus outlines of thirty randomly selected UK bachelors' degree courses in software engineering and found that only seven listed security-related modules. None of these seven courses included more than a single compulsory security related module (5.5% of course content), and in three of the seven cases such modules were entirely optional. With one exception, all the security related modules were offered in the final year, after development mindset and habits are likely to have been established by students for better or worse. We suggest that, at the very

least, basic security concepts should feature as an intrinsic component of all course modules from day one, rather than being isolated as a separate, and largely optional, advanced subject."

[5] Validity of accreditation, certification and attestation

In respect of accreditation (and hence in principle attestation) I quote from the *Integrated InfoSec submission to the 2023 UK DCMS call for views on software resilience and security for businesses and organisations*.

"Voluntary accreditation is in principle a good idea. It has worked well in other areas, as witnessed by the success of the BSI Kite Mark in raising and maintaining physical product standards. However the fundamental requirement for it to be effective is that assessment criteria are adequate to ensure that accreditation actually represents a high standard of performance. In the domain of physical goods this is relatively easy to accomplish. For example, a sample kite marked electrical plug can be physically examined and tested to determine whether it meets the accreditation criteria, and as it is mass produced that provides reasonable assurance that the sample is representative of the product, confirming that the production environment meets the required standards. In the case of software, it is commonly impractical to validate the production process robustly by examination of any one product. Inference from the good to the process will therefore be unreliable unless the good is found to be grossly deficient. This is a parallel to the network penetration test dilemma. A test that throws up problems can be trusted (at least as far as the identified problems are concerned), but a 'clean' test that shows no problems has no evidential value as it may simply mean that issues exist but have not been appropriately tested for.

Consequently, the only practical accreditation of organisations [seems at first sight to be] process-based. This throws up two key questions. First, even if a process is deemed to be 'best practice', is it possible to determine with confidence whether it consistently delivers product that meets requirements? Second, what exactly is best practice anyway?

The first of these problems is exemplified by ISO 9001 'quality' certification, which, for many organisations, consists of merely documenting what they do and continuing to do it, without significant regard for whether the results are actually good – it being deemed sufficient at audit that they are consistent. Hence the old joke about the ISO 9001 conforming toaster that incinerates every slice of bread to exactly the same charcoal brick. The only way to ensure that processes deliver to expectation is to set finite expectations as to outcomes, which brings us back to the problem of how to determine the quality of complex virtual entities such as software. Any sufficiently rigorous approach is likely to be very expensive and time consuming to conduct in individual cases.

The problem of what constitutes best practice is that it is primarily based on a consensus of current professional opinion. It is therefore very resistant to change even where its deficiencies become clear to some members of the relevant community. Change is only possible once a majority recognise the need. An exemplary parallel is the almost universal use in risk assessment of the 'risk matrix' which has survived as the tool of choice in many risk critical domains for half a century despite being objectively demonstrable as fundamentally flawed in that it yields almost meaningless results. Attempts by this respondent (and indeed not a few others) to draw attention to its deficiencies objectively from a mathematical perspective have not registered strongly, largely because the majority of the practitioner community does not currently approach risk assessment from that perspective.

In the context of software, best practice will be extremely hard to define exhaustively, as it must touch on both the lowest mechanical and highest conceptual aspects of the development process and everything in between, addressing at every level a huge range of possible hazards, not all of which have probably to date been identified.

So in summary, even supposing best practice can be reliably and exhaustively defined (which will be a major challenge), practicable accreditation against it can only be at the level of development processes, and validating whether product quality conforms to the expectations inspired by those processes will also be a serious challenge. The most adverse outcome of attempting prematurely to implement accreditation is that it could become a mere rubber stamp 'compliance' indicative of nothing concrete in terms of quality."

On developer certification I quote from my consultation response to the 2021 UK DCMS policy paper "*Digital Regulation: Driving growth and unlocking innovation*".

"While I specifically reject mandatory certification for all software developers, I do advocate a tiered system of certification. Where applications are not mission-, livelihood- or life critical, I see no need for certification, but the suggested 'kite mark' equivalent could be promoted as a commercial advantage to raise the quality of finished products. For mission critical applications, expectations should be set culturally, as in other engineering disciplines, that preferentially result in recruitment of certified developers. For livelihood- and life critical systems development, I would advocate mandatory certification.

Where mission-, livelihood- or life critical software is marketed to the public or produced for corporate clients or government agencies, the vendor or producer should be statutorily liable for faults and failures, with provision for adequate compensation. Regulation should preclude or nullify licensing terms that exclude liability or put barriers (e.g. contractual jurisdiction) in the way of claims.

As in other engineering disciplines, practitioners should be required to carry professional indemnity where they develop mission-, livelihood- or life critical systems other than as employees of an organisation that itself bears liability for product faults and failures."

I think the above quotes speak for themselves of the vast gulf between current and required expectations of verifiable competence, and clearly demonstrate that efforts must be made to close it by completely revising the way software developer education is conducted, validated and certified.

Considering these points, the problem of assuring accreditation and attestation validity will take a lot of solving, but implementing either while failing to achieve that assurance will lead to a false sense of security that is far more dangerous than unwitting lack of control over standards.

END OF DOCUMENT