Bonnes pratiques de cybersécurité pour le développement logiciel

Guidelines for secure software development

O. D'HENIN, F. SADMI, F. STOSSE
Bureau Veritas
60 Avenue du Général de Gaulle
92800 Puteaux
Olivier.dhenin@fr.bureauveritas.com
Franck.sadmi@fr.bureauveritas.com
Florian.stosse@fr.bureauveritas.com

Florent KIRCHNER
CEA-List
Nano-Innov Paris-Saclay
91120 Palaiseau
Florent.kirchner@cea.fr

Résumé

Avec l'émergence de la problématique « cybersécurité » dans le domaine industriel et l'importance croissante du développement logiciel pour les domaines de rupture (IoT, Industrie 4.0, nouvelles mobilités, etc.), Bureau Veritas, en partenariat avec le CEA-List a souhaité concentrer dans un guide unique l'ensemble des bonnes pratiques en matière de développement logiciel sécurisé pour l'industrie.

Cette démarche a été initiée par le retour d'expérience de nos clients bien souvent confrontés aux choix du référentiel à appliquer et à la pertinence des exigences elles-mêmes.

Ce guide s'adresse donc aux architectes du logiciel et aux développeurs d'applications industrielles. Il peut aussi servir de préconisations dans le cadre d'une sous-traitance.

Summary

With the emerging concerns about industrial cybersecurity and the ever-growing importance of software development for innovative fields (IoT, smart factory …), Bureau Veritas and its partner CEA-List have developed state of the art guidelines on best practices applied to secure software development for industrial users.

This approach was initiated by our customers' feedbacks about the difficulty to select and evaluate the relevance of existing guidelines.

Those guidelines are mainly intended for software architects and development teams, but can also be used as guidelines for suppliers.

1 Introduction

Le développement logiciel est aujourd'hui un facteur différenciant majeur pour toutes les industries. Ce constat peut être fait dans plusieurs secteurs :

- L'automobile, où les véhicules connectés sont devenus courants et où les constructeurs rivalisent pour proposer de nouvelles fonctionnalités aux clients finaux ;
- L'industrie lourde, où les lignes de productions deviennent connectées et pilotables à distance, à travers des fonctionnalités directement présentes dans les PLC (automates programmables) ou systèmes SCADA (systèmes de supervision);
- Le maritime, où les navires, mais aussi les systèmes portuaires (logistique, maintenance, …) font appels de plus en plus largement aux logiciels pour opérer ;

Si la maîtrise du développement logiciel est un atout et un levier de croissance important, la problématique de développer un logiciel robuste d'un point de vue Sûreté De Fonctionnement (SDF) et cybersécurité (et a fortiori dans des secteurs fortement contraints réglementairement, comme l'aéronautique ou le ferroviaire) peut vite devenir une gageure pour les industriels se lançant dans l'aventure.

Dans le domaine industriel, sur bien des aspects, les méthodes mises en œuvre pour la sûreté de fonctionnement servent foncièrement les objectifs de la cybersécurité. Ainsi le concept de défense en profondeur qui vient de la SDF s'applique parfaitement à la protection des systèmes face aux attaques informatiques. Cette approche recommande au concepteur

de produits (ou de systèmes) de superposer le maximum de couches de protection, notamment autour de la partie qui reste la plus visée : le logiciel.

D'autres activités issues de la sûreté de fonctionnement, comme la recherche des évènements redoutés, ainsi que la réalisation d'analyses préliminaires des risques et des dangers, sont autant d'éléments réutilisables dans le domaine de la cybersécurité.

Traditionnellement, ce type de protections de sécurité vient en complément et n'impacte pas directement la conception du logiciel. Il s'agit d'ailleurs – pour le moment – le plus souvent de contre-mesures mises en place dans une phase ultérieure à la conception du produit. La phase d'intégration système est en général la plus propice à ce genre d'actions. A titre d'exemples, on peut citer le durcissement des machines, une politique stricte en matière d'accès logique et physique ou encore le contrôle d'intégrité des données entrantes. La norme ISO/IEC 27002 est une aide précieuse dans ce domaine puisqu'elle va lister de façon quasi exhaustive ces moyens de protection et ce, quel que soit le domaine pour lequel est destiné le logiciel.

Mais lorsque l'attaquant a franchi l'ensemble de ces barrières, il tentera alors d'exploiter les failles du logiciel. La dernière défense est donc la résistance du logiciel lui-même et l'approche dite « Secure by design » va tenter d'y répondre, par l'application des principes suivants :

- 1er Principe : La réduction de l'exposition aux attaques en limitant par exemple ses interfaces qu'elles soient internes (Librairies, DLL…) ou externes (Réseau, IHM). Cette mesure reste la plus efficace et peut aller jusqu'au renoncement de certaines fonctionnalités du logiciel initialement prévues mais qui l'exposent trop fortement. On peut faire un parallèle avec le domaine de la SDF où une complexité logicielle trop importante sera proscrite (cf. les règles de développement pour la safety logicielle de la NASA proscrivant les fonctions trop longues, et les règles de développement SEI). En cybersécurité, un logiciel « compact » est de facto moins pourvu de failles de sécurité. C'est un principe souvent difficile à suivre, mais il doit être considéré dès les premières phases de conception. Son intérêt est primordial puisqu'il sert à la fois la cybersécurité et la SDF.

- 2ème Principe : Le logiciel doit intégrer ses propres moyens de défense comme par exemple la faculté de s'auto-vérifier afin de détecter toute perte d'intégrité du code exécuté. Ainsi dans une application distribuée, les différents modules logiciels peuvent s'échanger des signatures (par exemple basées sur une fonction de hachage à sens unique) du code exécutable selon un challenge idoine. Plus classiquement, l'échange de données sur un protocole chiffré est un autre exemple de moyen d'autodéfense du logiciel. Dans ces exemples, les intérêts de simplicité dictés par le domaine de la SDF vont être desservis par les principes d'ajout de fonctions de cybersécurité souvent complexes, il faudra donc peser les risques induits par les deux domaines et arbitrer au cas par cas.

- 3ème Principe : La confiance que l'on donne à un produit logiciel passe bien souvent par la confiance que l'on va prêter à son processus de développement. A ce stade, la Qualité, la SDF et la Cybersécurité vont s'unir et s'accorder parfaitement pour un objectif commun. Les phases de développement doivent donc s'inscrire dans un processus cadré visant à réduire autant que faire se peut les risques d'insertion de vulnérabilités (communément appelé SDLC, Security Development LifeCycle). On va retrouver ici les principes de développement suivants les méthodes toujours bien connues mais hélas pas toujours appliquées telles que : la gestion de configuration, les campagnes de tests boites banches, etc... Ces process de développement du logiciel sont largement déclinés dans des standards de qualité ou SDF (IEC 61508, DO178, ISO 26262, BV-SW-100). Mais il faudra également intégrer au processus de développement des phases spécifiques à la cybersécurité tel que la modélisation des flux associés et

leurs potentielles vulnérabilités : Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, Elevation of privilege (approche STRIDE de Microsoft).

2 Méthode

Le guide BV-SW-200 a été écrit en collaboration entre Bureau Veritas et le Laboratoire Sûreté et Sécurité du Logiciel du CEA-List, suite aux différentes observations et retours d'expérience que nous avons pu avoir dans le domaine industriel, où la SDF logicielle est déjà développée. Il adresse principalement le 3eme principe énoncé plus haut, en concentrant dans un volume réduit de trente pages, les bonnes pratiques issues de plusieurs standards en matière de développement sécurisé. Ce format facilite son adoption par les équipes de développement qui maitrisent déjà la SDF mais peu la cybersécurité. Ce guide dédié cybersécurité vient en complément des exigences de SDF du logiciel qui sont une base indispensable pour construire une démonstration de cybersécurité.

Il est présenté sous forme de liste d'exigences (appelées objectifs) réparties en deux niveaux : « basique » et « avancé ». Ces niveaux permettent de moduler l'approche en fonction des secteurs d'activités, du risque acceptable, du niveau d'investissement possible, mais également vis-à-vis de l'efficacité des couches de protections périmétriques évoquées plus haut. La façon de déterminer et d'allouer ce niveau de sécurité est laissée libre à l'utilisateur implémentant ce guide.

Chaque objectif est associé à des critères d'acceptation qui permettent l'évaluation du niveau de couverture. Cette conformité aux recommandations peut s'effectuer soit par autoévaluation soit par l'évaluation via un organisme externe. Un certificat peut alors être émis valorisant ainsi l'effort et les coûts engendrés par la prise en compte de la démarche cybersécurité.

Le guide adresse toutes les phases du développement du logiciel (de type cycle en V) ; la liste d'objectifs recoupe les phases du projet tel que : *SYStem, DESign, CHEcking, OPErations*.

Pour rappel, le cycle en V est un modèle conceptuel de développement, notamment utilisé dans le domaine du développement logiciel (mais il s'applique bien entendu à tout type de développement). Son nom est tiré de la représentation conceptuelle des différentes étapes du cycle :

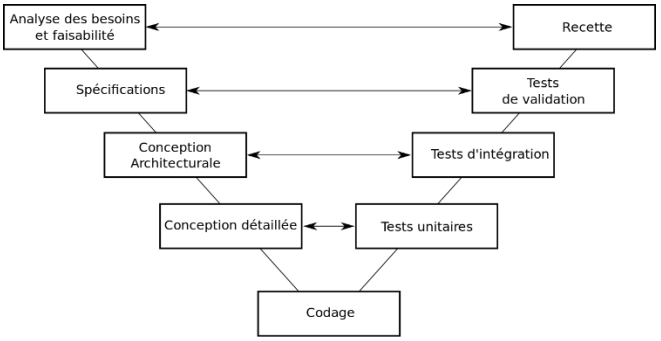

Figure 1 : Vue conceptuelle du cycle en V (crédits : Wikipédia)

La partie descendante du cycle concerne les différentes étapes d'expression et de recueil du besoin, d'établissement des spécifications et de raffinement du modèle de conception, tandis que la partie ascendante concerne les étapes de tests, vérification et validation.

Pour chacune de ces étapes, il est donc possible de les surcharger pour y intégrer des actions liées à la sécurité.

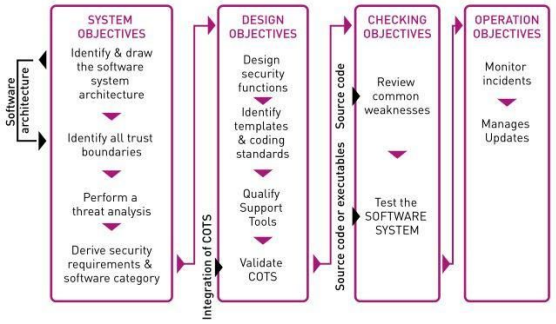

Figure 2 : Exemple de processus de sécurité pour un cycle de développement logiciel

2.1 Architecture logicielle

La partie *SYStem* va exprimer des exigences liées aux phases amont du développement logiciel, en demandant par exemple d'intégrer des étapes d'identification du périmètre de sécurité et de modélisation des menaces afin d'en dériver des exigences de sécurité applicables lors de la phase d'implémentation.

Dans cette partie, l'accent est mis sur la modélisation du système dans son contexte d'utilisation (qui peut se faire de façon embryonnaire sur une simple feuille de papier, ou de façon beaucoup plus formelle) et sur l'identification du périmètre du système ou composant logiciel, des acteurs (utilisateurs, mainteneurs, …), et de leurs échanges, à travers l'utilisation de diagramme de flux par exemple. Cette phase doit donc faire intervenir tous les acteurs impliqués dans le logiciel, et pas uniquement les experts sécurité et les développeurs. Le modèle pourra être ensuite raffiné de façon itérative et décomposé lui-même en sous modèles (selon le degré de complexité de l'application considérée).

Cette modélisation va permettre l'identification des frontières de confiance du système logiciel. Ces frontières désignent les processus d'échange ou de traitement des données au sein desquels le niveau de confiance (arbitraire) desdites données va changer. Cela peut par exemple concerner les entrées fournies par l'utilisateur, ou les échanges via des médias non sécurisés par défaut (Internet, etc.). C'est au niveau de ces frontières que l'effort de sécurisation devra être porté.

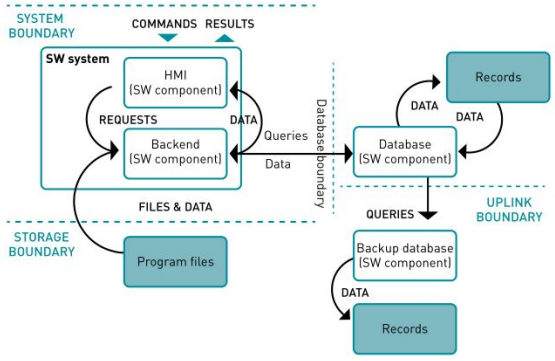

Figure 3 : Exemple de modélisation possible d'un système logiciel, avec identification des frontières de confiance (tirets bleus).

Pour pouvoir dériver des exigences de sécurité applicables aux différents flux et frontières, il faut dans un premier temps caractériser la menace. Il existe pour cela un certain nombre de méthodes et de métriques utilisables, telle que la méthode STRIDE de Microsoft, ou le classique C/I/D[1]. Une cotation du risque doit également être réalisée afin de pouvoir prioriser efficacement les actions à mener par la suite durant le développement.

[1] Confidentialité / Intégrité / Disponibilité

Pour chaque menace identifiée, des objectifs et des exigences de sécurité testables sont à formuler, en fonction de l'état de l'art.

Boundary	Link	S	T	R	I	D	E
System	User -->HMI	X		X		X	X
System	HMI--> User		X		X		
Database	Backend --> Database		X		X	X	
Database	Database --> Backend		X		X	X	
Uplink	Database--> Backup		X		X	X	
Storage	Program files --> Backend		X				

Figure 4 : Exemple d'application de la méthode STRIDE au système modélisé précédemment.

2.2 Design et conception

La partie *DESign* énonce pour sa part des règles et exigences applicables durant les phases de conception et d'implémentation du logiciel. Ces règles vont du style de programmation utilisé à la qualification des outils support, en passant par le bon usage des COTS[2].

Avant tout début de développement, il est important que l'équipe en charge formalise les règles de développement ainsi que les processus et outils permettant leur vérification. Ces règles peuvent être purement syntaxique, mais aussi contraignantes (constructions dangereuses interdites, etc.). De nombreuses organisations, notamment dans le monde de la défense, de l'automobile ou de l'aéronautique, ont publié des guides techniques à ce sujet, qui peuvent servir d'inspiration. Nous pouvons citer les règles de Secure Coding du CERT[3] -CC, les règles MISRA ou les règles de programmation utilisées chez Google.

La qualification et le bon usage des COTS sont également des points importants à étudier dans un développement logiciel, puisque ces derniers augmentent, parfois drastiquement, la surface d'attaque du système logiciel développé, et peuvent introduire également des vulnérabilités. Différents critères peuvent être utilisés pour évaluer la pertinence ou non d'en faire usage, tel que la disponibilité du code source, l'état de santé du projet (développement actif, nombre de contributeurs, suivi et maintenance, etc), la politique de sécurité associée, le nombre et le suivi de vulnérabilités connues, etc.

[2] Commercial Off The Shelf
[3] Computer Emergency Response Team

Un suivi particulier devra donc être fait suite à leur intégration, notamment au niveau des bulletins de sécurité émis par les CERT, pour pouvoir réagir rapidement en cas de nouvelle vulnérabilité trouvée dans un composant tierce partie.

Un point de vigilance important est à souligner concernant l'implémentation de fonctions de sécurité sensibles, telles que les fonctions de cryptographie si cette dernière est utilisée. Sauf pour les équipes disposant d'une expertise établie dans ce domaine, il est aujourd'hui fortement déconseillé d'implémenter directement ces fonctions. La bonne pratique serait en effet de se tourner vers des bibliothèques spécialisées (il en existe pour pratiquement tous les langages), qui disposent pour certaines d'entre elles de garanties de sécurité fortes (preuves formelles, audits externes, etc.). Ces bibliothèques fournissent en général des APIs et une documentation claire, déchargeant les développeurs des opérations (calcul en temps constant, remise à zéro des zones mémoire sensibles, etc.). Ces bibliothèques pouvant également être assimilées à des COTS dans certains cas, les mêmes points de vigilance sont à prendre en compte lors de leur usage.

2.3 Du bon usage des outils

Il existe aujourd'hui une myriade d'outils permettant aux développeurs de développer plus efficacement du code plus robuste.

Il est en effet plus facile de corriger un défaut logiciel pendant les phases de développement, quand la compréhension du code incriminé est encore présente, que pendant la phase d'exploitation opérationnelle du logiciel. Certains outils peuvent cependant demander une certaine expertise pour être pleinement exploitée, et il n'est pas inutile d'introduire l'usage de nouveaux outils de façon graduelle pour ne pas provoquer un rejet en bloc ou la non-utilisation de ceux-ci.

L'usage d'outils permettant l'analyse statique de code (c'est-à-dire sans exécution du programme), tels que les linters ou des passes de compilation spécifiques, permettent de détecter très tôt les erreurs potentielles. Ces outils n'offrent cependant pas tous les mêmes performances et garanties (couverture de code, type d'erreurs trouvées, exhaustivité, …)

Les linters sont un premier niveau d'outil, faciles de prise en main et d'utilisation. Ils permettent notamment de vérifier le respect des styles et règles de programmation choisis, et de repérer des erreurs triviales ou des constructions dangereuses.

Plus élaborés, les analyseurs statiques de code opérant par interprétation abstraite ou exécution symbolique sont également des alliés précieux pour mettre en évidence des bugs plus subtils. Tous les analyseurs n'offrent cependant pas les même garanties (soundness, faux positifs/négatifs, …) ni la même ergonomie d'usage, mais des solutions performantes sont aujourd'hui disponibles et intégrables aux produits d'intégration et/ou déploiement continu.

Les prouveurs automatiques ainsi que les assistants de preuves peuvent également être utilisés pour assurer un haut niveau de confiance via l'usage de méthodes formelles. Leur usage est cependant à envisager préférentiellement en début de développement, ou sur les portions critiques de code existant car leur mise en œuvre est plus complexe que les outils précédemment cités.

2.3.1 Sécurité à la compilation

Les compilateurs modernes sont aujourd'hui des aides importantes à l'intégration de fonctions de sécurité dans le binaire généré. La qualification et l'exploitation de chaînes de compilation récentes peuvent permettre des retours sur investissement en termes de sécurité non négligeables.

Nous pouvons citer :

- Position Independant Executable (PIE) : permet de tirer parti du mécanisme de sécurité ASLR[4] disponible sur les systèmes d'exploitation récent. Cette fonction permet de charger dans une plage mémoire aléatoire le corps du programme.
- Protections de la pile : les compilateurs peuvent introduire différentes protections permettant d'éviter un écrasement de la pile et du pointeur de retour de fonction.

[4] Address Space Layout Randomization

- Protection en lecture seule : l'usage de l'option RelRO permet de passer en lecture seule certaines zones mémoire (Global Offset Table, sections ELF, …) utilisables par un attaquant pour modifier le flux d'exécution du programme.
- Intégrité du flot de contrôle : une instrumentation du code à la compilation permet de vérifier à l'exécution que le graphe de flot de contrôle est bien respecté. Cela permet de détecter la modification par un attaquant du déroulé normal du programme.

Ces fonctionnalités ne sont cependant pas sans conséquences sur la taille du code ou sur la vitesse d'exécution de celui-ci.

2.3.2 Instrumentation du code à la compilation

En plus de permettre d'ajouter à moindre frais des fonctions de sécurité, les compilateurs modernes peuvent aussi fortement aider les développeurs à instrumenter leur code, de façon à repérer plus facilement et plus rapidement des défauts logiciels lors des phases de test.

C'est dans cette optique qu'une série de fonctions, appelées Sanitizers, a été rajoutée ces dernières années aux compilateurs phares du marché (GCC et Clang notamment).

Leur usage est cependant impérativement réservé aux builds de test, car leur coût en termes de performance est prohibitif en production, et que les informations fournies lors d'un crash peuvent être exploitées par un attaquant pour exploiter une faiblesse logicielle.

L'intérêt des sanitizers devient évident lorsque ces derniers sont combinés avec des outils de tests automatisés (fuzzing) assurant une bonne couverture de code, durant les phases de vérification et validation. En effet, les sanitizers vont permettre d'augmenter la couverture de code des outils de fuzzing, et de générer des rapports de crash plus détaillés sur l'origine du bug (adresses mémoires concernées, etc).

2.4 Vérification et validation

La partie *CHEcking* se concentre sur les exigences de sécurité relatives aux phases de test et de validation du cycle en V. Parmi celles-ci nous pouvons énoncer la recherche de vulnérabilités connues, comme les CWE[5] et CVE[6] , au sein du logiciel ainsi que dans ses dépendances (bibliothèques logicielles, binaires externes…). Il est aussi recommandé de réaliser des tests d'intrusion ou tests de fuzzing sur le produit pour vérifier la solidité des fonctions de sécurité implémentées.

L'intérêt des outils de vérification et de tests automatisés est ici également à souligner, surtout sur des bases de code dont la taille empêche de faire des revues manuelles exhaustives efficaces. Ils permettront en outre de détecter potentiellement des bugs ou des failles de sécurité dans des fonctions peu utilisées ou peu sollicitées à l'exécution.

2.5 Exploitation et maintien en condition de sécurité

La partie *OPErations* concerne la phase de mise en production et d'exploitation du logiciel, avec des exigences concernant la veille de vulnérabilités sur les dépendances externes prenant part à une fonction de sécurité (par exemple, le suivi des failles de sécurité d'OpenSSL pour les mécanismes cryptographiques) ainsi que la gestion des mises à jour et des incidents de sécurité.

L'usage de la télémétrie implémentée dans le logiciel peut permettre d'étudier l'usage ou le mésusage des mécanismes de protections internes ainsi que de la bonne mise en posture de sécurité en cas d'incident (fallback ou fail-safe). De la même façon, une documentation spécifique sur la configuration de sécurité ainsi qu'une procédure de retour aux paramètres par défaut peuvent également être fournis.

De plus, une procédure de mise à jour doit être définie et réalisée de manière sécurisée avec vérification de l'intégrité des patchs.

[5] Common Weakness Enumeration
[6] Common Vulnerabilities and Exposures

3 Conclusion

Comme nous avons pu le voir dans cet article, développer un logiciel de façon sécurisée passera forcément à la fois par des mesures techniques, mais également par des mesures organisationnelles.

Les mesures et solutions techniques sont aujourd'hui nombreuses et continuent de s'améliorer de jour en jour. Elles permettent de faciliter la vie des développeurs, de préparer au mieux un déploiement en améliorant et renforçant les phases de vérification et de validation, ou encore de mettre facilement à portée des fonctions et mécanismes de sécurité robustes.

Les mesures organisationnelles vont avoir pour but de créer une dynamique positive au sein des équipes de développement, et de renforcer les processus de contrôle de la qualité du code produit, permettant d'améliorer graduellement le niveau de sécurité du logiciel. Ces mesures impliquent notamment la formation au développement sécurisé, et le maintien des compétences spécifiques nécessaires à l'implémentation des mesures techniques.

Paru en juin 2017, le guide BV-SW-200 ambitionne donc d'offrir un état de l'art de ces mesures, en les associant avec des critères d'acceptation réalistes définis suites aux retours industriels que nous avons pu faire avec nos clients confrontés à cette problématique. Le guide est public et est librement téléchargeable à l'adresse suivante : http://www.bureauveritas.com/white-papers/cybersecurity-guidelines-for-development-and-assessment-bv-sw-200

Références

- NASA JPL Institutional Coding Standard for the C Programming Language
- SEI CERT C Coding standard
- IEC 62443, Security for Industrial Automation and Control Systems, IEC, 2009
- Guidelin&wVyBg6vGmp5NEV@NjkJ^U4Y$HgQ6Hes for Development & Assessment of Software, Bureau Veritas BV-SW-100, 2016
- Microsoft STRIDE: https://en.wikipedia.org/wiki/STRIDE_%28security%29
- MITRE CWE: https://cwe.mitre.org/
- MITRE CVE: https://cve.mitre.org/
- MISRA : https://www.misra.org.uk/
- CERT-CC Secure Coding : https://www.sei.cmu.edu/research-capabilities/cybersecurity/index.cfm
- Sanitizers : https://github.com/google/sanitizers
- Intégrité du flux de contrôle : https://clang.llvm.org/docs/ControlFlowIntegrity.html
- Position Independant Code/Executable : https://en.wikipedia.org/wiki/Position-independent_code
- ASLR : https://en.wikipedia.org/wiki/Address_space_layout_randomization
- Protection de la pile : https://en.wikipedia.org/wiki/Buffer_overflow_protection*
- Return-to-libc exploit : http://phrack.org/issues/58/4.html#article
- Fuzzing instrumenté : https://source.android.com/devices/tech/debug/fuzz-sanitize
- Génération de cas de tests aléatoires : https://en.wikipedia.org/wiki/QuickCheck

Mots clés

Cybersecurité, processus de développement, sécurité, logiciel sécurisé, bonnes pratiques.