Concevoir un système connecté pour la sécurité ET la sûreté

Un « bon » logiciel embarqué doit être conçu pour satisfaire à la fois des exigences de sécurité et de sûreté. Cependant, l'avènement de la connectivité induit des vulnérabilités dans les applications avec des conséquences sur le niveau de sûreté. Une compréhension profonde des deux concepts, associée à de bonnes pratiques de conception devient indispensable. Celles-ci portent sur l'adoption de normes de codage, l'utilisation d'outils d'analyse statique, les révisions de code et la modélisation des menaces, comme l'explique ici le Barr Group.

Le couplage étroit de la sûreté et de la sécurité, combiné à des niveaux de menace accrus, exige désormais que les développeurs comprennent parfaitement la différence entre la sécurité et la sûreté et appliquent les meilleures pratiques de l'industrie pour s'assurer que les deux sont intégrés dans tous les produits, et ce dès le début du cycle de conception (figure 1). Ainsi, avec l'IoT (Internet of Things), les systèmes sont dorénavant vulnérables via des actions à distance. Un cas récent impliquait par exemple des caméras de sécurité en réseau dans lesquelles on a trouvé des micrologiciels de type « cheval de Troie » (1), introduits via des ports logiciels ouverts. Ces ports sont utilisés par les pirates pour infecter les systèmes avec des logiciels malveillants, les « botnets », avant de lancer des attaques. Dans le cas des caméras, en l'occurrence des Sony, la firme japonaise a développé un patch de firmware que les utilisateurs peuvent télécharger pour fermer cet accès non protégé et se protéger contre de nouvelles attaques.

AUTEUR

Dan Smith, ingénieur principal, et Andrew Girson, CEO, Barr Group.

Autre exemple, pour alerter les développeurs, deux chercheurs ont volontairement piraté une automobile alors qu'elle était en mouvement, intervenant sur les fonctions du tableau de bord, de la direction, de la transmission et des freins (2). Le but de ce piratage non malveillant était de démontrer combien il était facile d'utiliser la connexion de l'opérateur réseau pour « s'introduire » dans la voiture. Il n'y pas eu d'accident, mais cette action de « hackage » a conduit tout de même au rappel de 4,1 millions de véhicules !

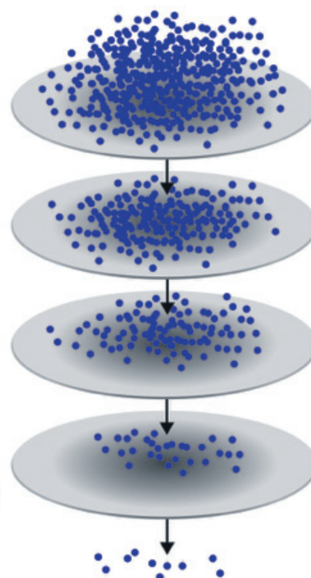
Des problèmes de conception…

Bien sûr, les systèmes n'ont pas eu à attendre d'être connectés à Internet pour être vulnérables et dangereux : un code embarqué mal écrit et de mauvaises décisions de conception ont déjà fait des ravages. Le cas des machines Therac-25, systèmes de radiothérapie lancés en 1983 pour traiter le cancer, est à cet égard symptomatique (3). Aujourd'hui, c'est devenu une étude de cas reconnue sur ce qu'il ne faut pas faire lors de la conception de système. Une combinaison d'erreurs logicielles, d'un manque de protection du matériel et d'une mauvaise prise de décision de conception a entraîné l'administration de doses fatales de rayonnement aux patients traités. Dans ce cas de figure, les problèmes portaient sur le processus de développement de logiciel immature et inadéquat (avec un logiciel non testable), une modélisation de la fiabilité et une analyse du mode de défaillance incomplètes, une absence d'examen (indépendant) des logiciels critiques et enfin une réutilisation incorrecte du logiciel à partir de modèles plus anciens.

Autre cas célèbre : en juin 1996, le vol 501 d'Ariane 5 a quitté son plan de vol prévu et s'est autodétruit car les contrôles de débordement de mémoire avaient été omis par les

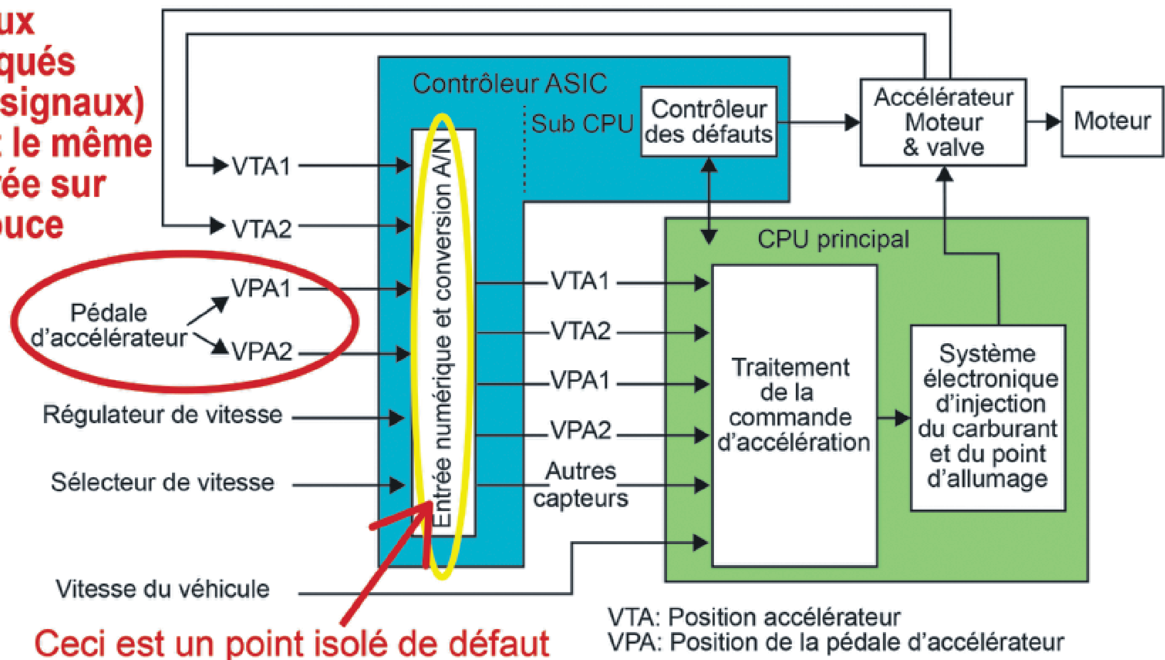
1 BONNES PRATIQUES DE CONCEPTION

Une bonne conception des logiciels et du matériel exige que plusieurs niveaux liés à l'assurance de la qualité, à la sûreté et à la protection soient utilisés tout au long du processus de conception.



2 POINTS DE DÉFAILLANCE D'UN SYSTÈME CRITIQUE
Les systèmes critiques pour la sécurité évitent les points isolés de défaut.

(Source: Prof. Phil Koopman4)



développeurs. Et lorsqu'une variable gérant la vitesse horizontale dépassait la valeur maximale prévue, il n'y avait aucun moyen de détecter ce débordement et d'y répondre de manière appropriée.
Et pourtant, malgré cet historique, du code critique et des vulnérabilités de sécurité continuent à ne pas être contrôlés. Un sondage conduit en 2017 par le Barr Group sur la sécurité et la sûreté des systèmes embarqués a ainsi révélé que, selon des ingénieurs travaillant sur des projets qui sont connectés à Internet et qui peuvent présenter un danger majeur s'ils sont piratés, la prise en compte de la sécurité dans le travail de conception est encore aléatoire. Ainsi, selon cette étude:
- 22 % des développeurs n'ont pas la sécurité comme exigence de conception;
- 19 % d'entre eux n'utilisent pas de règles de codage standard;
- 42 % ne conduisent pas d'opération de revue de code ou le font de manière occasionnelle;
- 48 % ne se donnent pas la peine de chiffrer leurs communications sur Internet;
- et plus d'un tiers ne font pas d'analyse statique.

Définir la sécurité et la sûreté

Face à ce constat, la compréhension de la véritable signification de la sécurité et de la sûreté est un bon début pour remédier à ces carences. Car les deux termes sont souvent confondus, et certains développeurs pensent à tort que s'ils écrivent un « bon » code, alors celui-ci sera à la fois sécurisé et sûr. Or, clairement, ce n'est pas le cas.
Un système «sûr» (dans le sens « sans risques ») est un système, qui, en fonctionnement normal, ne cause aucun dommage à l'utilisateur ou à toute autre personne. Un système dit à « sûreté critique » est un système qui potentiellement peut causer des blessures ou la mort en cas de dysfonctionnement. Dans ce cas, le but du concepteur est d'assurer que ce type de système ne soit jamais en dysfonctionnement ou ne tombe pas en panne.
En revanche, la sécurité concerne principalement la capacité d'un produit à mettre ses ressources à la disposition d'utilisateurs autorisés, tout en se protégeant des accès non autorisés, tels que les pirates informatiques. Ces ressources comprennent des données, du code et de la propriété intellectuelle (IP), des processeurs et des centres de contrôle système, des ports de communication, de la mémoire et du stockage de données statiques.
Ces termes étant établis, il apparaît clairement que si un système peut être sécurisé, il n'est pas automatiquement « sûr » : un système mal conçu peut être très sécurisé, aussi bien qu'un système pensé pour être sûr et sans risques. Cependant, à l'inverse, un système peu sécurisé est toujours dangereux, car même s'il est fonctionnellement sûr par conception, sa vulnérabilité aux accès non autorisés peut le rendre dangereux à tout moment.

Concevoir pour la sécurité et pour la sûreté

En ce qui concerne la conception pour la sûreté, il existe de nombreux facteurs à considérer, comme le montre l'exemple du Therac-25, mais dans le cadre de cet article nous allons nous intéresser plus particulièrement au problème du firmware.
Un bon exemple d'application critique est l'automobile, avec des codes qui peuvent atteindre plus de 100 millions de lignes pour des uti-

lisateurs finaux novices ou distraits : les conducteurs ! Parallèlement, la tendance est à l'ajout de fonctionnalités et de codes de sécurité supplémentaires, liés à l'utilisation de caméras et de capteurs, d'un côté, et de l'autre à l'intégration de communications entre véhicules et infrastructures fixes (liaisons V2I) et entre véhicules (liaisons V2V). Avec comme conséquences, à nouveau une augmentation exponentielle de la quantité de code embarqué dans les voitures et à nouveau une extension des zones, rendant les opérations de débogage de plus en plus ardues. Cependant une partie du temps consacré à la vérification de ces logiciels embarqués peut être réduit en suivant certains principes fondamentaux :

- réaliser un partitionnement matériel/logiciel optimisé, opération qui influe directement sur les performances temps réel, la sûreté, la fiabilité et la sécurité ;
- mettre en place de régions de confinement des pannes ;
- éviter les points de défaillance isolés (figure 2) (4) ;
- gérer les exceptions causées par des erreurs de codage, le programme lui-même, la gestion de la mémoire ou des interruptions parasites ;
- inclure des contrôles de débordement mémoire ;
- contrôler les données provenant du monde extérieur et pouvant être contaminées (vérification des limites d'utilisation et du CRC, Cyclical Redundancy Checking) ;
- tester à tous les niveaux (test composant, test d'intégration, test système…).

Pour la sécurité, un concepteur ou un développeur doit aussi se familiariser avec les subtilités de l'authentification des utilisateurs et des dispositifs, de l'infrastructure de clé publique (PKI) et du cryptage des données. En plus de rendre les ressources disponibles pour les utilisateurs autorisés et de les protéger contre les accès non autorisés, la sécurité signifie également qu'un système ne fait pas de choses inattendues ou dangereuses face à une attaque ou à un dysfonctionnement. Certes, les attaques interviennent sous diverses formes, y compris le déni de service simple (DoS, Denial of Service) ou distribué (DDoS, Distributed DoS). Mais, bien que les développeurs ne puissent pas contrôler ce qui attaque le système, ils peuvent contrôler comment le système réagit à l'attaque. Dans ce cadre ils doivent prendre conscience que la façon de réagir s'applique à l'ensemble du système, car un système est aussi sécurisé que son lien le plus faible et il est certain que l'attaquant trouvera ce lien. Un exemple de cible particulièrement faible est la mise à jour du firmware, avec l'activation de la fonctionnalité dite de Remote Firmware Update (RFU) du dispositif. Comme elle peut facilement être attaquée, il est donc judicieux d'avoir mis en

3 NORME DE CODAGE MISRA

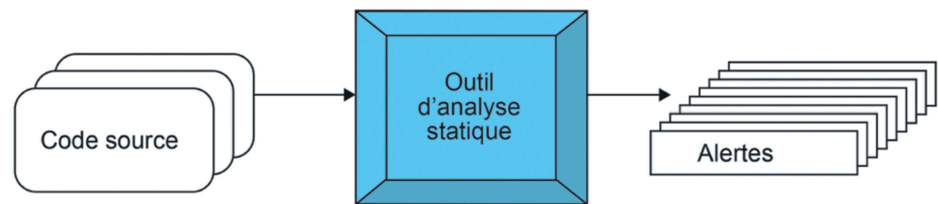
Les avantages de l'adoption de normes de codage comme MISRA vont au-delà de l'aide pour éviter les bogues ; cela rend également le code plus lisible, cohérent et portable.

Processus de développement	Niveau d'intégrité				
	0	1	2	3	4
Spécification et conception	ISO 9001	Méthode structurée.	Méthode structurée supportée par l'outil CASE	Spécification formelle de ces fonctions à ce niveau	Spécification formelle du système complet. Génération automatisée du code (si disponible)
Langages et compilateurs		Langage structure standardisé.	Sous-ensemble restreint d'un langage structure standardisé. Compilateurs validés ou testés (si disponible)	Idem 2.	Compilateurs indépendamment certifiés avec une syntaxe et une sémantique formelles prouvées (si disponible)
Gestion de la configuration produits		Tous les produits logiciels. Code source.	Relations entre tous les produits logiciels. Tous les outils	Idem 2.	Idem 2.
Gestion de la configuration processus		Identification unique. Produit en accord avec la documentation. Contrôle d'accès. Changement d'autorisation.	Changements liés au contrôle et à l'audit. Processus de confirmation.	Changement automatisé et construction du contrôle. Processus de confirmation automatisée	Idem 2.

place une stratégie adéquate. Comme par exemple demander à l'utilisateur soit de désactiver le RFU, soit de charger une mise à jour qui nécessite uniquement des images successives faisant alors office de signature numérique.

4 ANALYSE STATIQUE

Les outils d'analyse statique exécutent une «simulation» du code source, analysent la syntaxe et la logique, et génèrent des alertes au lieu de fichiers objet.



Il est à noter que dans l'analyse de la sécurité, la cryptographie est rarement le maillon le plus faible. Au contraire, les attaquants vont chercher ailleurs des surfaces d'attaque rendues vulnérables en raison de la manière dont le code est implanté, de la sécurité des protocoles, des API, ainsi que des attaques via un canal parallèle. Et la quantité d'effort, de temps et de ressources investis dans chacun de ces domaines dépend du type de menace de sécurité, chacun ayant des défenses spécifiques. Voici quelques mesures générales qu'un développeur peut prendre pour réduire la vulnérabilité du produit:

- utiliser un microcontrôleur sans mémoire externe;
- désactiver l'interface JTAG;
- mettre en place un démarrage sécurisé;
- utiliser une clé principale pour générer la clé spécifique de chaque unité du dispositif;
- utiliser l'obfuscation du code objet;
- implanter l'autotest à la mise sous tension (POST, Power-On, Self Test) et l'autotest intégré (BIST, Built-In Self Test).

A propos de l'obfuscation (méthode générale visant à obscurcir le sens d'un code pour le rendre difficilement compréhensible au premier abord), il existe une école de pensée qui enseigne « la sécurité par l›obscurité ». Mais cette façon de faire peut être fatale si on se repose exclusivement sur elle. Car chaque secret crée un point potentiel de défaillance (5). Tôt ou tard, les secrets sont révélés, que ce soit par l'intermédiaire de l'ingénierie sociale, des employés mécontents ou par des techniques telles que le dumping et l'ingénierie inversée.

Des outils et méthodes incontournables

Au-delà, même si de nombreuses techniques et technologies peuvent aider les développeurs et les concepteurs à atteindre un niveau élevé de fiabilité et de sécurité, il existe quelques étapes fondamentales qui garantissent qu'un système est déjà optimisé en termes de sécurité et de sûreté.

La première consiste à concevoir en faisant appel à des standards de l'industrie et des normes spécifiques aux applications. Il s'agit notamment des normes MISRA et MISRA-C, ISO 26262, Automotive Open System Architecture (Autosar), CEI 60335 et CEI 60730. L'adoption d'une norme de codage comme MISRA contribue ici non seulement à éviter les bogues, mais à rendre le code plus lisible, cohérent et portable (figure 3).

La seconde consiste à utiliser l'analyse statique (figure 4). Il s'agit ici d'étudier un code sans avoir à l'exécuter via une exécution symbolique, en d'autres termes via une une simulation. Bien que l'analyse statique ne soit pas la panacée, elle ajoute un niveau supplémentaire d'assurance, car elle est efficace pour détecter les bogues potentiels, comme l'utilisation de variables non initialisées, la possibilité de débordement d'entier de type overflow/underflow et le mélange de types de données signés et non signés.

La troisième porte sur la révision de code. Cette opération améliorera l'exactitude du code tout en facilitant la maintenabilité et l'extensibilité. Les révisions de code sont également utiles dans les cas de rappels et/ou de réparations sous garantie avec les problèmes de responsabilité qui en découlent.

Enfin, quatrième action importante, la mise en place de modèles de menace. Pour de faire, il est utile de démarrer en commençant par l'utilisation d'un arbre d'attaque en se mettant à la place d'un attaquant. Dans cette approche, il faut identifier les objectifs d'attaque (chaque attaque a un arbre séparé) et pour chaque arbre il faut déterminer les différentes attaques probables et identifier les étapes et les options pour chacune d'entre elles.

Faire ces quatre étapes de base pour minimiser les erreurs et améliorer la sécurité et la sûreté paraît simple, mais elles prennent du temps, de sorte qu'un développeur doit budgétiser en conséquence ces opérations, en essayant d'être le plus réaliste possible. Par exemple, ajouter entre 15% et 50% de temps de conception en plus pour la révision de code paraît raisonnable. Bien que certains systèmes nécessitent des révisions de code complètes, alors que d'autres non. A ce niveau les outils d'analyse statique peuvent nécessiter des dizaines, voire des centaines d'heures de travail pour la mise en place initiale. Mais une fois ce travail réalisé, lorsque le processus de développement démarre, aucun temps supplémentaire n'est nécessaire, ce qui finit par s'avérer rentable. ■

(1) Comptes backdoor trouvés dans 80 modèles de caméras de sécurité Sony IP (Backdoor accounts found in 80 Sony IP security camera models) http://www.pcworld.com/article/3147311/security/backdoor-accounts-found-in-80-sony-ip-security-camera-models.html

(2) Après le piratage des Jeep, Chrysler rappelle 1,4M de véhicules pour la correction des bogues (After Jeep Hack, Chrysler Recalls 1.4M Vehicles For Bug Fix) https://www.wired.com/2015/07/jeep-hack-chrysler-recalls-1-4m-vehicles-bug-fix/

(3) Tué par une machine : le Therac-25 (Killed By A Machine: The Therac-25) http://hackaday.com/2015/10/26/killed-by-a-machine-the-therac-25/

(4) Une étude de cas sur l'accélération involontaire des Toyota et la sécurité logicielle (A Case Study of Toyota Unintended Acceleration and Software Safety)https://users.ece.cmu.edu/~koopman/pubs/koopman14_toyota_ua_slides.pdf

(5) Citation de Charles Mann, paraphrasant Bruce Schneier (Atlantic Monthly, septembre 2002) (Quote by Charles Mann, paraphrasing Bruce Schneier (Atlantic Monthly, Sept. 2002.) tre figure 1.