A propos de moi : Je suis Valentin Lefils, doctorant à l'Université Lille 1 où je prépare une thèse sur la sécurité des systèmes embarqués. Je travaille au sein de l'équipe 2XS, que j'ai rejoint en octobre dernier. Avant cela j'ai obtenu un Master en informatique à l'université de Lille 1, au cours duquel j'ai eu notamment l'occasion de m'intéresser à l'analyse statique ainsi qu'à la sécurité des systèmes embarqués.

Tissage de code pour le suivi d'exécution de binaire destinés à l'informatique embarquée

Aujourd'hui la majorité des processeurs équipant l'informatique embarquée inclut des mécanismes de renforcement contre l'exploitation de faille. Cependant les applications restent vulnérables face à des classes d'attaques par détournement de flot sur du code légitime (*Return to libc, JOP[1], ROP[2]*) qui permettent de prendre le contrôle d'applications même en présence de mécanismes matériels (*NX bit*) et/ou logiciels (*ASLR*)[3].

L'objectif des travaux présentés est donc d'apporter des garanties logicielles en renforçant les programmes lors de la compilation. Une famille de défenses s'appuyant sur l'intégrité du flot de contrôle (*Control Flow Integrity* ou *CFI*) a fait son apparition [4] afin d'apporter des garanties sur l'exécution attendue d'un programme et donc de contrer les attaques détournant le flot de contrôle. Abadi défini le CFI de la façon suivante : *"La politique de sécurité CFI implique que l'exécution d'un programme doit suivre un chemin validé par le graphe de flot de contrôle déterminé avant exécution".* Divers travaux proposant des implémentations de CFI ont été proposés, mais celles-ci sont en général spécialisées pour contrer certains types d'attaques connus ou pour protéger certains aspects de l'exécution[5] (*jump, return, function pointer call*).

Un des principaux challenges scientifiques dans ce contexte est de mettre en place une surveillance du flot d'exécution aussi complète que possible tout en tenant compte des contraintes imposées par les systèmes embarqués. Coudray et al. [6] ont proposé une politique de CFI complet appelée PICON. Elle repose sur une vérification intégrale des transitions entre Blocs de Base (BB) selon une stratégie : modèle / trace / vérification. L'extraction du graphe de flot de contrôle (CFG), est réalisée par analyse statique. Le code est ensuite instrumenté afin de produire une trace de son exécution dynamiquement pour qu'elle puisse être comparée en temps réel au modèle attendu par une unité dédiée en charge de stopper l'exécution si celle-ci n'est pas conforme au graphe de flot de contrôle.

Cependant PICON présente deux faiblesses principales. Premièrement cette approche permet de détecter un flot d'exécution incorrect mais ne permet pas de détecter une pause ou un détours dans l'exécution. Par exemple en l'absence de *NX bit*, toute injection de code aurait pour effet de ne plus remonter de traces au moniteur (le code injecté n'étant pas instrumenté), celui-ci resterait donc en attente de la suite de la trace sans produire d'erreur. Le code instrumenté peut également être la cible d'une attaque de type *ROP*, dans ce cas de figure le programme attaqué se contenterai de produire des traces d'exécution sans

jamais revenir dans le code d'origine. Il est intéressant de noter que cette attaque permettrait de passer entièrement sous silence certaines parties du code et donc de sauter d'une partie du CFG à une autre (et ainsi exécuter n'importe quel code présent en simulant une série de transition légitimes).
Deuxièmement leur implémentation utilise un moniteur interne (un processus séparé créé au lancement du programme), alors que dans le contexte de l'informatique embarquée il est préférable d'externaliser les traitements afin que ceux-ci ne pèsent pas plus que nécessaire sur le système embarqué ciblé. De plus afin de limiter au maximum le coût de communication, il serait préférable que la trace produite soit d'abord agrégée dans tampon puis envoyée pour analyse au moniteur (via le réseau par exemple).

La solution que nous proposons ajoute une vérification sur le temps d'exécution de chaque BB. Le code est instrumenté lors de la compilation afin d'ajouter des *timestamps* dans la trace produite durant l'exécution (à l'entrée et à la sortie de chaque BB) afin de pouvoir vérifier plusieurs propriétés.

La première propriété à vérifier est l'instantanéité des transitions. Lors du retour d'une fonction A vers une fonction B par exemple, si une seconde entière s'écoule entre la fin du dernier BB de A et le début du BB de B suivant le call, alors il existe une possibilité que le *return* soit corrompu et qu'une 3ème fonction non instrumentée ait été exécutée avant de revenir dans la fonction B (ou que la communication avec le moniteur à été interrompue pendant le détournement de flot avant de reprendre dans la fonction B). Pour ce faire le moniteur possède un mode d'apprentissage où il enregistre les temps observés d'exécution de chaque BB afin de déterminer un pire temps d'exécution qui servira de seuil pour déclencher les alertes. De plus, afin de garantir la présence de temps de référence pour chaque BB, une couverture de code aussi exhaustive que possible au regard du temps d'exécution doit être réalisée lors de la phase d'apprentissage.

La seconde propriété est de s'assurer que chaque BB s'exécute dans un temps attendu basé sur le temps d'exécution maximum observé au préalable. De cette manière une attaque simulant un chemin légitime avec une *ROP chain* (et donc ayant des temps d'exécution de BB presque nuls) a une probabilité plus importante d'être détectée.

Cette approche est principalement valide dans le milieu des systèmes embarqués qui offrent une meilleure garantie de régularité dans les temps d'exécution que dans les systèmes classiques notamment en raison de la faible interférence des caches (souvent absents), de l'absence d'accès concurrent (les architectures multi-coeur n'étant pas la norme) et de l'absence de *swap* disque (l'exécution en flash NOR étant privilégiée au *swap*).

La trace générée dans le tampon fini par le remplir. Le code généré déclenche alors la transmission du paquet vers le moniteur sans interrompre l'exécution du programme en utilisant un *DMA* (accès direct à la mémoire). A la réception de ce paquet, le moniteur vérifie l'exactitude et la durée des transitions entre BB, le temps d'exécution de chaque BB et la somme des temps d'exécution des BB qui doit être cohérente avec le délai depuis le dernier paquet reçu (ce qui permet de détecter si la trace a été modifiée au moment de la

communication entre le programme et le moniteur ou si la trace a été falsifiée par un programme corrompu souhaitant simuler un comportement légitime).

Pour évaluer notre approche, nous utilisons deux cartes STM32F7 équipées d'un processeur ARM Cortex-M7. La première carte est utilisée comme cible, du code instrumenté est chargé dessus puis exécuté. Elle envoie ensuite la trace de son exécution à une deuxième carte contenant le code du moniteur qui se charge de l'analyser et de remonter les éventuelles alertes. Notons que dans un premier temps nous nous intéressons à un programme simple, exécuté directement sur le matériel sans système d'exploitation (ce type de programme est caractéristique des petits systèmes embarqués).

Les principaux enjeux de cette phase d'évaluation sont la réduction du surcoût en temps d'exécution induit par notre solution, la mise en place d'exemples concrets d'attaques ainsi que la recherche de vulnérabilités encore exploitables.

[1] Bletsch, Tyler, Bletsch Tyler, Jiang Xuxian, Vince W. Freeh, and Liang Zhenkai. 2011. "Jump-Oriented Programming." In Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security - ASIACCS '11. doi:10.1145/1966913.1966919.

[2] Roemer, Ryan, Roemer Ryan, Buchanan Erik, Shacham Hovav, and Savage Stefan. 2012. "Return-Oriented Programming." ACM Transactions on Information and System Security 15 (1): 1–34.

[3] A Short Guide on ARM exploitation : https://www.exploit-db.com/docs/24493.pdf

[4] Abadi, Martín, Abadi Martín, Budiu Mihai, Erlingsson Úlfar, and Ligatti Jay. 2005. "Control-Flow Integrity." In Proceedings of the 12th ACM Conference on Computer and Communications Security - CCS '05.

[5] Goktas, Enes, Goktas Enes, Athanasopoulos Elias, Bos Herbert, and Portokalidis Georgios. 2014. "Out of Control: Overcoming Control-Flow Integrity." In 2014 IEEE Symposium on Security and Privacy.

[6]Coudray, Thomas, Arnaud Fontaine, and Pierre Chifflier. "Picon: Control Flow Integrity on LLVM IR." In SSTIC 2015.