Étude d'un outil d'analyse: `angr`

Philippe Pépos Petitclerc

2020-04-30

Introduction

L'outil choisi pour la présente étude est `angr`. C'est une suite d'outils et de librairies qui vise tnat l'automatisation de l'exploitation de programmes binaires que l'assistance à l'analyse manuelle. Afin de bien comprendre l'outil, on presentera le milieu dans lequel vit `angr` et à d'où viennent les problématiques qui sont adressés. On detaillera ensuite ce que l'outil tente d'accomplir. On survolera également les limites d'un outil comme `angr`. Nous explorerons ensuite comment utiliser `angr` pour accomplir certaines tâches.

Contexte

`angr` est né d'une candidature au *Cyber Grand Challenge*, une compétition d'exploitation automatique de programmes binaires. On cherche donc à analyser des programmes binaires, en langage machine, et donc sans le code source. Analyser un programme en langage machine induit plusieurs complexités additionnelles par rapport à analyser un programme dont on à accès au code source. Par exemple, l'identification des octets de donnée par rapport aux octets de code, les informations de types n'est plus présente, et bien d'autres.

Aussi, la compétition ayant un accent sur l'exploitation automatique, `angr` présente un fort axe offensif. Il cherche donc principalement à aider ou automatiser des tâches offensives comme la rétroingénierie, la recherche de vulnérabilités et leur exploitation.

Objectifs et solutions

`angr` propose deux façons différentes de l'utiliser. Il se présente comme une librairie d'analyse de programmes binaires (voire un cadriciel) ainsi qu'une suite de d'outils d'analyse. L'approche la plus flexible est d'utiliser `angr` comme une librairie et c'est ce que nous étudierons.

Le cadriciel expose donc différentes analyses qui assistent à la compréhension d'un programme ou à sa rétroingénierie automatique. Par exemple, on peut

construire une graphe de flot de contrôle d'un programme en utilisant différentes stratégies. Il est également possible de faire l'exécution symbolique (dynamique) d'un programme ou d'une section d'un programme, avec ou sans *Veritesting* (Avgerinos et al. 2014), avec ou sans recours à l'exécution concrète. Il y a une panopelie d'autres analyses que l'on peut trouver dans le répertoire `angr/analyses` du dépôt de code du projet ("Angr/Angr" 2020).

Une des fonctionnalités intéressante pour un cadriciel d'analyse est sa capacité à passer de l'information d'une analyse à une autre. `angr` utilise une base de connaissance à l'interne qui permet de traîner de l'information sur le programme analysé et de la rendre disponible à chaque analyse ou à un analyste utilisant son API.

De plus, `angr` expose des services pour interagir ou questionner avec un executable binaire comme une abstraction du chargeur dynamique ou des fonctions utilitaires pour trouver les adresses des fonctions, extraire des métadonnées ou des symboles, etc.

Dans plusieurs des analyses d'`angr`, il est requis de faire appel à un solveur SMT. Il y a donc également une abstraction vers le solveur qui simplifie son utilisation et son intégration avec l'état de la mémoire d'un programme.

En ligne avec la compétition du *Cyber Grand Challenge*, `angr` offre également plusieurs fonctionnalités qui permettent d'automatiser une partie de la tâche de rétroingénierie d'un programme et son exploitation. `angr` permet notamment de construire des graphes de flot de contrôle, de dépendences de données et d'extraire les valeurs potentielles de la mémoire. Plusieurs techniques d'exploration (découverte des différentes branches d'exécution possibles) sont également exposés. Du côté de l'exploitation, `angr` offre des fonctionnalités de génération d'exploits de programmation orientée-retour (*ROP*) mais surtout, l'API du solveur SMT simplifie la recherche de tampons où il est possible d'écrire un payload et la recherche d'états qui satisfont des contraintes d'exploits (le compteur ordinal contient une adresse où l'on peut écrire un *payload*, par exemple).

Limitations

`angr` agglomère, en quelque sorte, les limitations de tous ses composantes. Les faiblesses habituelles de chacune des analyses disponibles dans `angr` s'appliquent également au travers de l'outil. Les problématiques classiques d'explosion combinatoire des exécuteurs dans le cas de l'exécution symbolique dynamique, par exemple, sont tout à fait présentes. On retrouve également des problèmes avec certaines instructions dont la bibliothèque de *lifting* (*VEX*, qui vient du projet *Valgrind*), n'a pas encore d'implémentation en représentation intermédiaire. Les procédures complexes à analyser sont généralement simulées dans `angr`. En quelque sorte, c'est un avantage car on peut maintenant analyser une partie de leur comportement, mais cette simplification induit très certainement de l'erreur dans certains cas d'utilisation de ces procédures. Aussi, certaines procédures ne

sont tout simplement pas encore implémentés en version simulée (il est toujours possible de le faire en tant qu'analyste). Les appels systèmes sont un bon exemple de ces situations.

Méthodologie et exploration

Afin de démontrer l'exploration qui a été faite de l'outil, on montre quatres exemples disponibles dans le dépôt de code associé à cette recherche (Petitclerc 2020). Les exemples couvrent la construction de graphes de flots de contrôle, l'exploration automatique de programmes, l'ajout de contraintes manuelles pour la recheche d'états particuliers ainsi qu'une démonstration d'exploitation automatique tirée du dépôt d'exemples("Angr/Angr-Doc" 2020).

Le premier exemple est disponible sous la forme d'un *IPython Notebook* dans le repertoire `notebooks` du dépôt et se nomme `CFG Recovery - fauxware.ipynb`. On y montre comment reconstruire le graphe de flots de contrôle du programme `fauxware`. Une librairie tiers permet de générer une représentation graphique.

Le second, nommé `Exploration - fauxware.ipynb` montre comment utiliser les mécanismes d'exploration de l'espace d'états pour trouver les valeurs d'entrées à donner au programme pour exécuter les différentes branches possibles du programme `fauxware`.

Dans le troisième exemple, disponible en tant que `Contraintes Manuelles - fake.ipynb`, on contraint manuellement l'état ciblé en forçant un tampon à respecter le format d'un *flag*. On tente ensuite de résoudre le système de contraintes afin d'obtenir un *flag* valide.

Le dernier exemple est plutôt différent, on explore les capacités de recherche de vulnérabilités et d'exploitation automatique d'`angr`. On peut trouver le programme vulnérable et le script pour générer l'exploit dans le répertoire `bin/aeg`. Dans ce script, on voit comment trouver un état vulnérable en modélisant les prérequis à l'exploitabilité sous la forme de contraintes. On cherche donc un état où il est possible que le compteur ordinal contienne une valeur symbolique. Si c'est le cas, on cherche si cet état a une zone mémoire symbolique de longueur suffisante pour contenir le *shellcode*. Le cas échéant, on tente de résoudre les entrées nécessaires pour que le compteur ordinal contienne l'adresse du tampon contenant le *shellcode*. Si on trouve une solution, cette solution représentera les entrées à donner au programme pour exécuter le *shellcode* injecté.

Conclusion

Le moteur et cadriciel d'analyse de programme binaires qu'est `angr` est impressionnant par son efficacité et sa flexibilité. Il est possible d'accomplir ou d'accélérer l'analyse de programmes binaires de plusieurs façons en l'utilisant. L'option de modéliser les états exploitables des programmes via des contraintes

sur la mémoire et les registres pi de chercher à résoudre ces états est particulièrement intéressant. Par contren, bien qu'à l'avant-garde de la recherche dans le milieu de l'exploitation automatique de programmes binaires, `angr` est susceptible de souffrir des limitations habituelles des analyses qu'il utilise. Il en revient à l'analyste de comprendre ces situations et soit d'utiliser les composantes qui contournent ces limitations soit d'éviter d'analyser les morceaux de programmes problématiques. Pour aller plus loin, il serait intéressant d'étendre la modélisation des états exploitables à d'autres types de vulnérabilités comme des vulnérabilités de format de chaînes qui sont certainement plus complexes à exploiter.

Bibliographie

"Angr/Angr." 2020. angr. https://github.com/angr/angr.

"Angr/Angr-Doc." 2020. angr. https://github.com/angr/angr-doc.

Avgerinos, Thanassis, Alexandre Rebert, Sang Kil Cha, and David Brumley. 2014. "Enhancing Symbolic Execution with Veritesting." In *Proceedings of the 36th International Conference on Software Engineering*, 1083–94. ICSE 2014. Hyderabad, India: Association for Computing Machinery. https://doi.org/10.1145/2568225.2568293.

Petitclerc, Philippe Pépos. 2020. "Ppepos/Inf889a_angr." *GitHub*. https://github.com/ppepos/inf889a_angr.