Retour d'expérience: Enseignement de la "*Sécurité logicielle*" à l'Université de Bordeaux

Emmanuel Fleury
Université de Bordeaux, France.
<emmanuel.fleury@u-bordeaux.fr>

***Résumé*—Cet article présente un retour d'expérience sur un ensemble d'UEs qui ont pour but de former les étudiants à la sécurité logicielle au sein du Master "Cryptologie et Sécurité Informatique" (Master CSI) de l'Université de Bordeaux. Ce cursus a été développé entre 2005 et 2020 et continue à évoluer en fonction des différents besoins du domaine.**

I. Contexte du Master CSI de Bordeaux

Le Master "Cryptologie et Sécurité Informatique" (CSI) de l'Université de Bordeaux a été crée sous l'impulsion de Christine Bachoc en 1998, d'abord sous la forme d'un DESS, puis en tant que Master lors du passage au LMD dans les années 2000. Ce Master a la particularité de se trouver à cheval entre le département de Mathématiques et le département d'Informatique pour offrir une conjonction des deux domaines. De fait, son cursus comporte une part égale de cryptologie et de sécurité informatique et les étudiants peuvent colorer leur parcours plus vers l'un ou l'autre des domaines en fonction des options qu'ils prennent au fur et à mesure des semestres. Le pari que nous avons fait, en forçant cette interdisciplinarité, est que la sécurité informatique dans toutes ses formes nécessitera de plus en plus de comprendre des notions de cryptologie. Enfin, notons aussi que nous avons obtenu le label SecNumEdu de l'ANSSI en janvier 2017.

Les débouchés de ce Master sont assez variés mais touchent très souvent le domaine de la recherche, ce qui est le but visé. La figure 1 présente les débouchés recensés sur un questionnaire que nous avons fait passer aux promotions de 2005 à 2019, nous avons obtenu 105 réponses sur environs 300 étudiants diplômés sur cette même période (entre 20 et 30 étudiants diplômés par an).

Le premier semestre de notre cursus (voir figure 2, p.2) sert essentiellement à aplanir les différences entre les populations d'étudiants qui viennent de Mathématiques ou d'Informatique. Il sert aussi à leur donner des bases communes pour suivre les cours spécialisés qui commencent réellement à partir du second semestre. Au troisième semestre, les étudiants peuvent se constituer une spécialisation soit vers la sécurité informatique, soit vers la cryptologie en fonction de leurs choix d'options. Notez que, comme l'étudiant doit constituer son semestre en choisissant 5 UEs parmi 8 et que nous avons fait en sorte d'en avoir 4 en cryptologie et 4 en sécurité informatique, son cursus devra inévitablement comporter une UE dans le domaine qu'il n'a pas choisi afin d'équilibrer la formation qu'il reçoit. Le Master1 compte environs 30 à 40 étudiants et le Master2 environs 20 à 30 étudiants en moyenne.

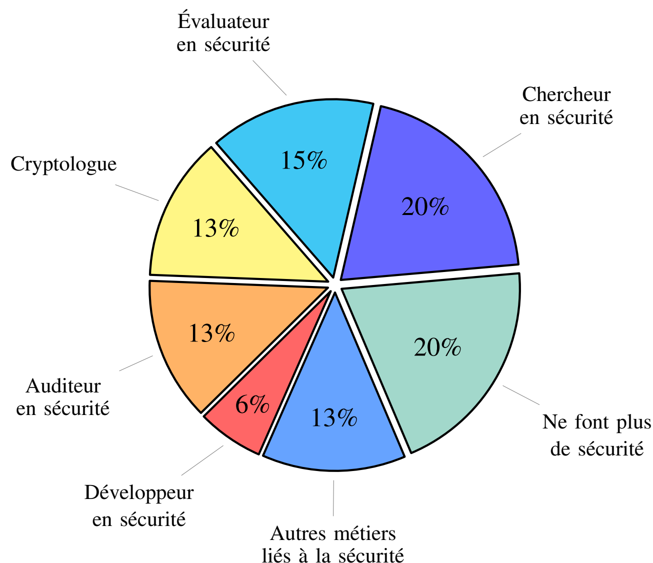

FIGURE 1. Débouchés du Master Cryptologie et Sécurité Informatique.

Enfin, pour le stage, nous essayons de faire en sorte qu'il donne lieu à la rédaction d'un mémoire de Master sur un sujet théorique ou technique qui soit suffisamment avancé. Cependant, nous avons bien souvent du mal à négocier ce genre de sujets avec les entreprises qui préfèrent considérer les stages comme des périodes de pré-embauche pour étudiants et non comme des stages académiques.

II. L'enseignement de la sécurité logicielle

L'enseignement de la sécurité logicielle au sein du Master se fait avec pour but de former des gens capables de réaliser des audits de sécurité sur des logiciels soit à partir des sources, soit seulement à partir du binaire. Le parti pris a été de focaliser une bonne partie de l'enseignement sur les techniques offensives d'exploitation et de rétro-ingénierie logicielle en donnant aux étudiants la possibilité d'acquérir une expérience pratique sur des exercices relativement réalistes (et souvent empruntés à des CTF d'un niveau "*correct*"). Évidemment, ces objectifs ont été affinés au cours des années grâce aux recommandations des

Modules de première année (M1)

Semestre 7	Mises à niveau en Info. & Math.	Théorie de la complexité	Arithmétique	Programmation	Théorie de l'information	Option
	0 ECTS	6 ECTS	6 ECTS	6 ECTS	6 ECTS	6 ECTS

Mise à niveau en informatique : 8h de cours intégrés la première semaine sur Unix, la programmation (C, Java) et LaTeX.
Mise à niveau en mathématiques : 8h de cours intégrés la première semaine sur les bases de l'arithmétique et de l'algèbre.
Options semestre 7 : Système d'exploitation ou Analyse, classification, indexation des données.

Semestre 8	Cryptologie	Sécurité logicielle	Calcul formel	Option	Projet	Anglais	Stage
	6 ECTS	6 ECTS	6 ECTS	6 ECTS	3 ECTS	3 ECTS	0 ECTS

Options semestre 8 : Administration réseau, Méthodes formelles, Outils Hilbertiens, Programmation parallèle.

Modules de seconde année (M2)

Semestre 9	Cryptologie avancée	Cryptanalyse	Courbes elliptiques	Algorithmique arithmétique	Sécurité réseau*	Sécurité système	Cartes à puce	Vérification des logiciels
	6 ECTS	6 ECTS	6 ECTS	6 ECTS	6 ECTS	6 ECTS	6 ECTS	6 ECTS

Semestre 9 : Choisir 5 UEs parmi les 8. Notez que '*Sécurité réseau*' requiert '*Administration réseau*' du semestre 8.

Semestre 10	Projet	Séminaires	Stage / Mémoire de recherche
	6 ECTS	0 ECTS	24 ECTS

Semestre 10 : Les séminaires sont donnés (entre janvier et février) par des intervenants extérieurs.

FIGURE 2. Cursus du Master Cryptologie et Sécurité Informatique de l'Université de Bordeaux.

encadrants de stage que nous rencontrions et du retour de nos anciens étudiants une fois insérés dans le milieu professionnel.

Pour atteindre ce but, nous nous reposons sur une séquence de trois UEs de 6 ECTS (12 semaines avec 2h de cours et 2h de TD sur machine par semaine) chacune étalées sur les semestres 7 à 9 du Master avec pour idée directrice que chaque cours est le pré-requis du précédent pour finalement amener les étudiants au niveau visé. Nous débutons par un cours de programmation C/Java (Semestre 7, Master1), puis nous passons à un cours consacré à la sécurité des applications en espace utilisateur (Semestre 8, Master1), et nous concluons avec un dernier cours sur la programmation et l'exploitation de vulnérabilités en espace noyau (Semestre 9, Master2).

III. Semestre 7 : Programmation C/Java

Le but de ce cours est de donner des notions de développement logiciel ainsi que de permettre aux étudiants de comprendre le fonctionnement des programmes et du système Unix. Le cours est séparé en deux parties, les 6 premières semaines sur la programmation C, puis les 6 dernières semaines sur la programmation Java.

La partie programmation C est structurée autour d'un projet individuel identique pour tous les étudiants. Le projet est guidé par cinq devoirs à rendre au rythme d'un par semaine, puis d'un projet qui consiste à partir de la base des cinq devoirs et d'amener le logiciel un peu plus loin que les devoirs. Les logiciels rendus à l'issue du projet sont généralement de l'ordre de 1000-1500 lignes de code et peuvent être relativement élaborés algorithmiquement.

Les projets qui sont utilisés pour ce cours ne changent pas tous les ans car il est nécessaire de maîtriser l'ordre d'arrivée des notions et d'avoir une complexité croissante et contrôlée au fur et à mesure de l'avancée du cours. C'est pour cela que nous faisons une implémentation complète des projets que l'on propose avant de donner les sujets aux étudiants la première fois. Pour l'instant nous alternons entre un solveur de Sudoku généralisé qui gère des grilles de 1×1 à 64×64 ou bien un joueur de Reversi avec des plateaux de tailles de 2×2 à 10×10. Le parti pris est de toujours prendre des problèmes ayant une complexité au moins NP (P-space pour Reversi) et de s'appuyer sur des structures de données non-triviales qui forcent à des manipulations bit-à-bit (*preemptive-sets* [1] pour le Sudoku, *bitboard* [2] pour le Reversi).

L'UE impose un rendu de devoir chaque semaine, ils sont donnés sous la forme d'une feuille qui introduit de manière informelle la spécification d'un ensemble de structures de données et de fonctions à implémenter pour la semaine à venir. Une fois rendu, chaque devoir est évalué en attribuant 10 points à des tests de conformité à la spécification et 10 points à la relecture de code par le chargé de TD.

Pour la partie "tests", chaque "*fail*" retranche un point sur les 10. Le programme étant construit au fil des cinq devoirs, le nombre de tests augmente jusqu'à 300–400 tests pour le projet final. C'est à la charge des étudiants de gérer leur code et de corriger les tests qu'ils ont raté pour l'échéance suivante.

La lecture de code, consiste en une relecture rapide du code des étudiants par le chargé de TD. Cette phase permet de s'assurer que les *coding-styles* imposés sont suivis (chose assez difficile à faire comprendre aux étudiants au début) et traquent toutes les mauvaises habitudes de programmation qui sont difficilement détectées automatiquement sans faux positifs (non vérification du code de retour de certaines fonctions, code maladroit, complexité excessive, duplication de code, ...)

À l'issue de ces cinq devoirs, les étudiants se retrouvent à la tête d'une base de code qu'ils doivent maîtriser suffisamment pour en améliorer la fiabilité, les performances, voire l'architecture à condition que les tests continuent à passer. Généralement, nous demandons un incrément sur l'efficacité du logiciel face au problème qu'il doit résoudre et nous testons les performances de chacun des logiciels en les comparant entre eux (nous mettons systématiquement celui que nous avons programmé dans l'évaluation pour être en parfaite transparence avec les étudiants sur notre niveau de maîtrise de la programmation par rapport au leur. Il arrive régulièrement que des étudiants nous dépassent, ce qui, en général, nous motive pour reprendre et améliorer notre code). Dans le projet de Reversi, nous faisons un affrontement global entre tous les joueurs artificiels qui ont été programmés. Dans le projet du solveur de Sudoku, nous avons crée un ensemble assez large de grilles de complexité croissantes qui nous permettent d'évaluer les stratégies adoptées par les programmes.

La partie Java est plus classique et se focalise surtout sur les aspects "génie logiciel" et les mécanismes liés à la programmation objet. Elle permet aux étudiants de souffler un peu après l'effort qu'ils ont fourni pour le projet de C. L'évaluation finale est un examen sur machine pour lequel nous utilisons des sujets du type "*Google Code Jam*" [1].

Le plus gros défaut de ce cours est qu'il passe très mal à l'échelle sur un grand nombre d'étudiants. Il est néanmoins envisageable de recourir à une plate-forme en ligne pour automatiser une bonne partie du travail sur le rendu des devoirs et l'application des tests sur les projets. Il est aussi envisageable de proposer une aide à la relecture de code. Mais, jusqu'à présent aucune plate-forme testée n'a été totalement satisfaisante, ni n'a réellement prouvée qu'elle permettait une amélioration des choses. Il faut aussi signaler que le nombre des projets disponibles reste encore trop faible, nous n'avons actuellement que deux projet et nous aimerions le porter à quatre pour nous assurer que les promotions ayant des projets similaires sont suffisamment éloignées dans le temps.

Au final, ce que les étudiants retirent de ce cours est très varié [2] mais, le but est essentiellement de les mettre en position de toucher du doigt le fonctionnement d'un programme sur un système Unix, de se rendre compte du processus de création d'un logiciel, de manipuler des outils tels que `gcc`, `gdb` `valgrind` et d'autres encore, ainsi que de les confronter à des bugs qu'ils ont eux-mêmes commis. En effet, pour bien comprendre la sécurité logicielle, il faut savoir se mettre à la place de celui qui a conçu le programme que l'on évalue. Cette étape pose donc les bases sur lesquelles nous allons pouvoir nous appuyer pour la suite.

IV. Semestre 8 : Sécurité des applications

Le cours du semestre suivant est consacré à l'exploitation de failles classiques dans un programme en C/C++. Une partie du cours est aussi consacrée à la rétro-ingénierie et aux techniques d'obfuscation classiques. Les sujets abordés sont les suivants :

1) Introduction to software security
2) Usual programming flaws
3) x86 assembly
4) *ARM assembly** (projet)
5) Obfuscation and reverse-engineering
6) Shellcodes
7) Stack buffer-overflow exploitation
8) Advanced stack buffer-overflow exploitation
9) *Heap buffer-overflow exploitation** (projet)
10) Format strings and other stories
11) *Fuzzing techniques** (projet)
12) *Metasploit** (projet)

Nous commençons par explorer les failles qui ne nécessitent aucune connaissance de l'assembleur pour être exploitées (injection de commandes, *race conditions*, attaques `LD_PRELOAD`, ... Pour cela nous nous basons essentiellement sur les challenges de la machine virtuelle Nebula du site "*exploit-exercises* [3]". Puis, nous abordons l'assembleur x86 (32-bit et 64-bit) avec leurs ABIs "systemV" respectives et la notion d'appels système directement depuis l'assembleur.

Puis, nous présentons la partie sur la rétro-ingénierie logicielle et l'obfuscation via un ensemble de crackmes qui illustrent chacun une technique particulière d'obfuscation ou d'anti-debug en expliquant comment marche la technique et comment la contourner efficacement. Nous utilisons soit `gdb`, soit `radare2` (ou Cutter), soit Ghidra (très pratique avec son décompilateur intégré). Malheureusement, nous n'avons pas suffisamment de temps à consacrer à cette partie pour aborder les techniques d'obfuscation profondes qui nécessiteraient un vrai cours de compilation au préalable.

Pour étendre un peu la pratique de l'assembleur, nous avons eu l'idée de mettre l'étude des shellcodes juste après avoir vu l'assembleur. Ce qui permet de consolider les connaissances des étudiants dans ce domaine.

Vient ensuite, une partie assez longue sur l'exploitation des *buffer-overflows* sur la pile, où nous commençons par désactiver toutes les sécurités avant de les réactiver les unes après les autres (ASLR, NX/DEP, RELRO, PIE, ...). Dans cette partie, nous parcourons les articles classiques qui expliquent les diverses techniques d'exploitation pour les failles de ce type [3]–[6]. Nous passons un peu plus de temps sur les ROPs

1. Voir : https://codingcompetitions.withgoogle.com/
2. Voir : https://cryptologie.net/article/15/sudoku-solver/ et https://cryptologie.net/article/24/done/
3. Voir : https://exploit-exercises.lains.space/

que sur les autres techniques (un TD entier à pratiquer) car il s'agit d'un concept qui est souvent utilisé en réalité.

Enfin, nous notre traitons des problèmes de *format-string* [7] essaye d'être exhaustif et nous permet de voir plusieurs attaques, notamment par des réécritures d'une adresse sur la GOT qui sont des attaques fréquemment utilisées pour contourner certaines contre-mesures.

Il reste encore des parties qui sont à créer ou à développer, notamment les parties concernants les attaques sur la *heap* et l'utilisation avancée du framework Metasploit qui restent encore très faibles, à notre humble avis, et il s'agira très probablement d'un point que nous développerons dans les années qui suivront avec la mise en place d'exercices adaptés.

L'évaluation de cette UE se fait via un projet par binômes qui consiste à valider un maximum de challenges Root-me [4] dans les catégories App-system et Cracking, la validation finale se fait via un rapport qui explique la solution de chacun des challenges validés (et si l'explication n'est pas convaincante, les points ne sont pas mis). Toutefois, comme Root-me est une plate-forme publique et que les challenges risquent d'avoir des solutions qui fuitent sur Internet, il nous a fallu penser à un système de notation robuste à cet effet. Pour cela, nous décernons un nombre de points à chaque challenge en fonction du nombre de groupes de la classe qui l'a résolu, ainsi on attribue (un script calcule la note maximale du groupe régulièrement pour qu'ils sachent où ils en sont) :

- 1 point si moins de 10% des groupes l'ont résolu.
- .75 point si moins de 50% des groupes l'ont résolu.
- .5 point si moins de 90% des groupes l'ont résolu.
- .25 point si moins de 100% des groupes l'ont résolu.
- .125 point si 100% des groupes l'ont résolu.

Assez étrangement, bien que la collusion globale soit la stratégie optimale dans ce cas, les étudiants adoptent rapidement des stratégies individualistes et entrent en compétition pour résoudre le plus grand nombre de challenges possibles. Ce qui est exactement le but recherché dans notre cas.

Enfin, nous concluons cette UE par un examen de 3 heures sur table pendant lesquelles il est demandé aux étudiants de lire un article de recherche non vu en cours et de répondre à des questions s'y référant. Les archives des examens se trouvent sur ce dépôt Github [5] maintenu par un ancien étudiant.

Cette UE est encore grandement améliorable et nous aimerions pouvoir développer certains aspects à l'avenir. Notamment, comme nous l'avons signalé précédemment, la partie sur l'exploitation de la *heap* est encore à adapter correctement. De plus, il serait vraiment très intéressant d'élargir le cours en enseignant aussi l'assembleur ARM et transposer toutes les techniques qui s'appliquent sur le x86 à l'architecture ARM. Enfin, la partie sur la rétro-ingénierie pourrait, elle aussi, être développée mais il faudra trouver le moyen de trouver un cours de compilation orienté vers les sujets qui nous intéressent qui puisse être placé avant ce cours.

4. Voir https://www.root-me.org/
5. Voir : https://github.com/mpgn/astudiaeth/

V. SEMESTRE 9 : SÉCURITÉ DES SYSTÈMES

Le dernier cours se place en premier semestre de Master2, il est donc optionnel. En conséquence, seuls des étudiants volontaires le rejoignent. Ce qui permet de compter un peu plus sur leur motivation à apprendre des choses dans le domaine. Il s'agit aussi du cours qui est le plus "*jeune*", il n'a que 3 ans, mais le contour reste assez clairement de s'aventurer au-delà de l'espace-utilisateur en espace-noyau voire d'explorer la sécurité physique.

La première partie du cours consiste à aborder la programmation de modules Linux plutôt orienté vers les pilotes logiciels. Nous présentons et mettons en place tous les outils pour permettre une compilation du noyau Linux, la manipulation des différentes options à travers `kconfig`, savoir compiler/charger/décharger des modules, *etc.*

La deuxième partie consiste à réaliser un ensemble de fonctionnalités propres aux rootkits noyau : Hooking sur la table des appels systèmes, faire disparaître le module rootkit de la liste des modules, création d'une backdoor, keylogger noyau, cacher des fichiers préfixés par une certaine chaîne de caractères, cacher les processus d'un utilisateur particulier, *etc.* Ces exercices nous permettent d'approfondir les connaissances du noyau par les étudiants tout en utilisant le prétexte du rootkit pour leur faire manipuler différentes APIs du noyau.

La troisième partie consiste à introduire artificiellement des modules ayant des bugs et à les exploiter depuis l'espace-utilisateur. Nous voyons les techniques du NULL-dereference et les différentes techniques d'exploitation d'overflow dans la pile (avec les techniques de reconstruction de "*trap-frame*" pour contrôler le retour via `iret`). Les protections SMEP et SMAP sont aussi abordées et comment les contourner en forgeant un ROP spécifique, éventuellement en se servant d'un *stack-pivot* pour basculer ensuite en espace-utilisateur une fois le SMEP désactivé [8]. Nous nous appuyons sur les challenges de type LinKern proposés sur la plate-forme Root-me qui sont assez bien réalisés même si nous pensons bientôt être capable de reproduire ces exercices en salle machine sur le compte des étudiants sans dépendre d'une plate-forme extérieure.

Le contrôle continu consiste à faire réaliser aux étudiants une présentation (30mn) suivie d'un TD (1h) sur des sujets liés à la sécurité des systèmes. Ce format permet de choisir des sujets variés, qui suivent l'actualité et de faire réaliser aux étudiants qu'il est parfois difficile d'écrire de bons exercices pour les autres. Enfin, l'examen final est calqué sur le principe du cours du semestre précédent (lecture d'article pendant 3h).

Ce cours nécessite encore pas mal de travail, notamment au niveau d'un approfondissement du cours de programmation noyau et des différentes techniques d'exploitation noyau. Enfin, la partie "*attaque physique*" n'est pas encore finalisée car il nous manque encore quelques compétences dans le domaine et aussi pas mal de matériel (analyseurs logiques, plate-formes d'attaques, *etc.*). Néanmoins, nous sommes de plus en plus persuadés que ce genre de contenu sera un plus indéniable pour notre formation.

VI. Conclusion et perspectives

Globalement, hormis les connaissances que nous abordons, nous essayons de présenter aux étudiants la sécurité comme une "*attitude*" face à un ensemble de problèmes et de souligner l'importance de garder un esprit analytique capable de comprendre les choses dans toute leur profondeur sans rien laisser échapper et, donc, d'en détruire toute la "*magie*", car c'est souvent là que se cachent les failles. Ceci surtout pour transmettre un peu la curiosité des "*hackers*" originels tels qu'ils étaient avant d'être dévoyés par le grand public.

Aussi, l'utilisation d'articles (académiques ou pas) pendant le cours est un point qui nous permet d'ancrer la sécurité dans le monde universitaire et la recherche, là où la plupart des gens dissocient totalement les deux mondes.

En outre, après presque 15 ans à essayer de nous former et d'améliorer cette formation, il reste encore malheureusement tant à faire qu'il nous semble difficile de prédire combien de temps sera nécessaire encore pour atteindre un niveau satisfaisant sur cette formation. Le facteur le plus limitant étant que le nombre d'enseignants de haut niveau en sécurité semble être à l'heure actuelle encore trop restreint, ou trop fragmenté sur le territoire, pour espérer atteindre une masse critique suffisante pour faire émerger des formations qui se détachent vraiment des autres. Il serait peut-être bon de favoriser les échanges ou les projets conjoints entre les formations, voire intensifier la formation des enseignant eux-mêmes.

Enfin, nous aimerions particulièrement remercier nos collègues de Mathématiques dont les enseignements forment particulièrement bien à la rigueur, chose absolument nécessaire lorsqu'on aborde des questions de sécurité informatique. Nous pensons réellement qu'une bonne partie du succès (relatif) de cette formation est due à leurs cours.

Références

[1] J. F. Crook, "A pencil-and-paper algorithm for solving sudoku puzzles," *Notices of the AMS*, vol. 56, no. 4, pp. 460–468, april 2009.

[2] C. Browne, "Bitboard methods for games," *ICGA Journal*, vol. 37, no. 2, pp. 67–84, 2014.

[3] AlephOne, "Smashing the stack for fun and profit," *Phrack*, vol. 49, 1996.

[4] klog, "The frame pointer overwrite," *Phrack*, vol. 55, 1999.

[5] Nergal, "The advanced return-into-lib(c) exploits : Pax case study," *Phrack*, vol. 58, 2001.

[6] H. Shacham, "The geometry of innocent flesh on the bone : return-into-libc without function calls (on the x86)," in *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*, 2007, pp. 552–561.

[7] Scut, "Exploiting format string vulnerabilities," March 2001.

[8] V. Nikolenko, "Practical smep bypass techniques on linux," 2016.

[9] S. Bratus, M. E. Locasto, M. L. Patterson, L. Sassaman, and A. Shubina, "Exploit programming : From buffer overflows to "weird machines" and theory of computation." *;login :*, vol. 36, no. 6, 2011.