Hybrid Engine for Polymorphic Shellcode Detection

Udo Payer

udo.payer@iaik.at

Peter Teufl

peter.teufl@iaik.at

Mario Lamberger

Mario.lamberger@iaik.at

http://www.iaik.at

Overview

- POSITIF Project
- Shellcodes/Polymorphic shellcodes
- Proposed Detection Engine
- Results
- Conclusions/Outlook

POSITIF

- (*Policy-based Security Tools and Framework*) is funded by the European Commission
- main goal is to design *automatic tools* to support *security managers* in protecting *networked infrastructures* and *applications*
- ideas and solutions developed by POSITIF will be available as *open-source*

http://www.positif.org/ipartners.html

Shellcodes

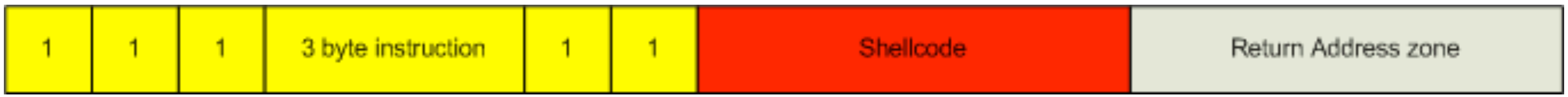
- Exploit buffer overflows to inject malicious code
- Typically consist of three zones: *NOP zone, shellcode, return address zone*
- Can be detected by simple signatures
- Invention of polymorphism (also used for viruses)
- shellcodes without NOP zones

Shellcode Detection

- *NOP zone*: IDS search for repeating 0x90 patterns
- *Shellcode*: IDS search for shellcode patterns (e.g. /bin/bash)
- *Return address zone*: IDS search for return addresses of known buffer overflows (e.g. Buttercup)

Polymorphic Shellcodes

- *NOP zone*:
  - Detection of pure 0x90 NOP zones is simple
  - Use other instructions than 0x90 (NOP)
  - Not every instruction can be used
  - All one byte instructions can be used safely
  - n-byte (n>1) instructions decrease probability of jumping into aligned code



Polymorphic Shellcodes

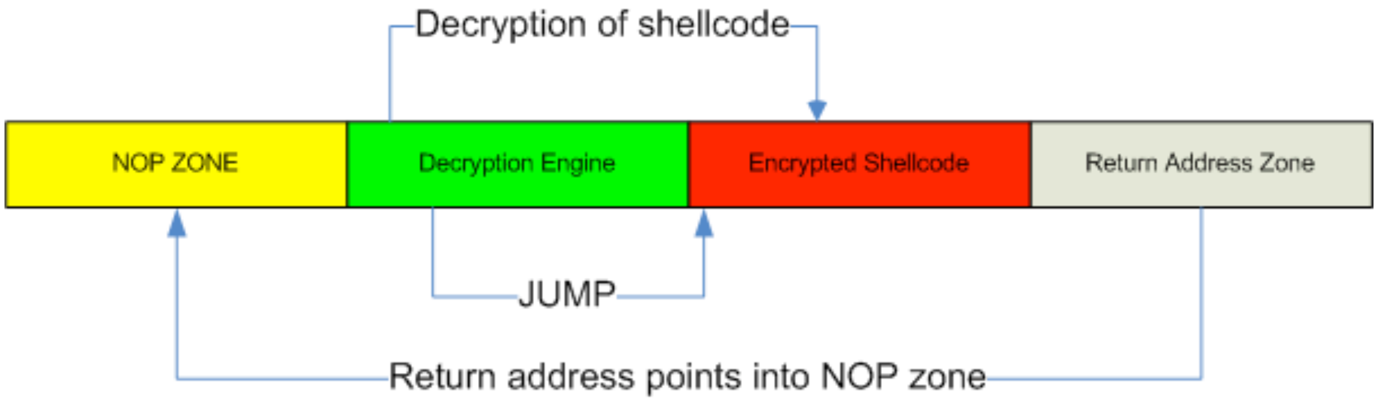
- *Shellcode*:
  - Signatures can be derived: e.g. search for /bin/bash
  - Encryption of shellcode (simple algorithms are enough): e.g. xor encryption
  - Mutation of encryption engine:
    - insert junk instructions
    - use other functions to achieve same result (e.g. ***push data***, ***pop reg*** instead of ***mov reg,data***)

Polymorphic Shellcodes

- *Return address zone*:
  - Cannot be encrypted
  - Mutation of least significant byte
  - Buttercup detection method

Polymorphic Shellcodes

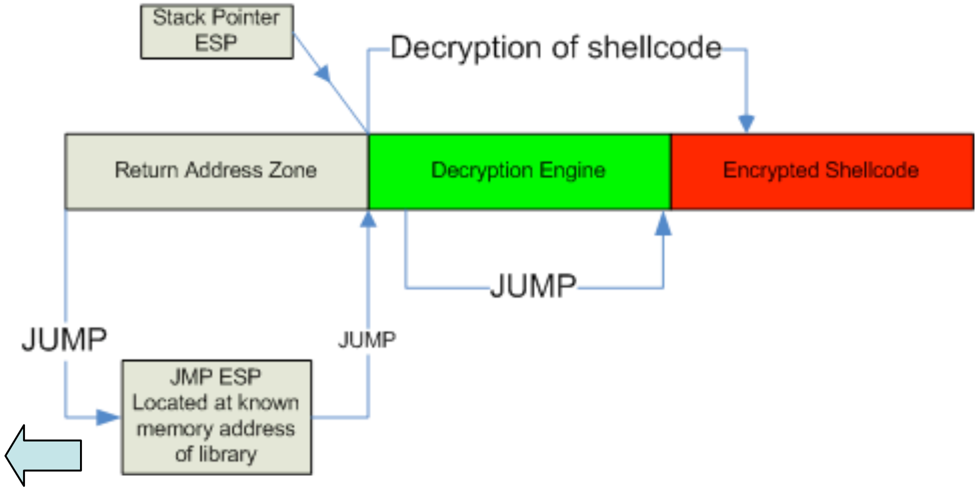
NOP ZONE
2E inc edi
2F inc esp
30 inc ebp
31 push esi
DECRYPTION ENGINE
32 jmp short 0x67
34 pop eax
35 xor edx,edx
37 mov dl,0x20
39 mov ecx,[eax]
44 rol ecx,0xb
47 add ecx,0xc29e092f
4D xor ecx,0x5ffde9d7
58 sub eax,0xfffffffe
5D inc eax
5F sub dl,0x3
63 jz 0x6c
65 jmp short 0x39
67 call 0x34
ENCRYPTED SHELLCODE
6c xxxxxxx



Shellcode with NOP zone (JMP ESP)

Polymorphic Shellcodes

Shellcode without NOP zone (JMP ESP)



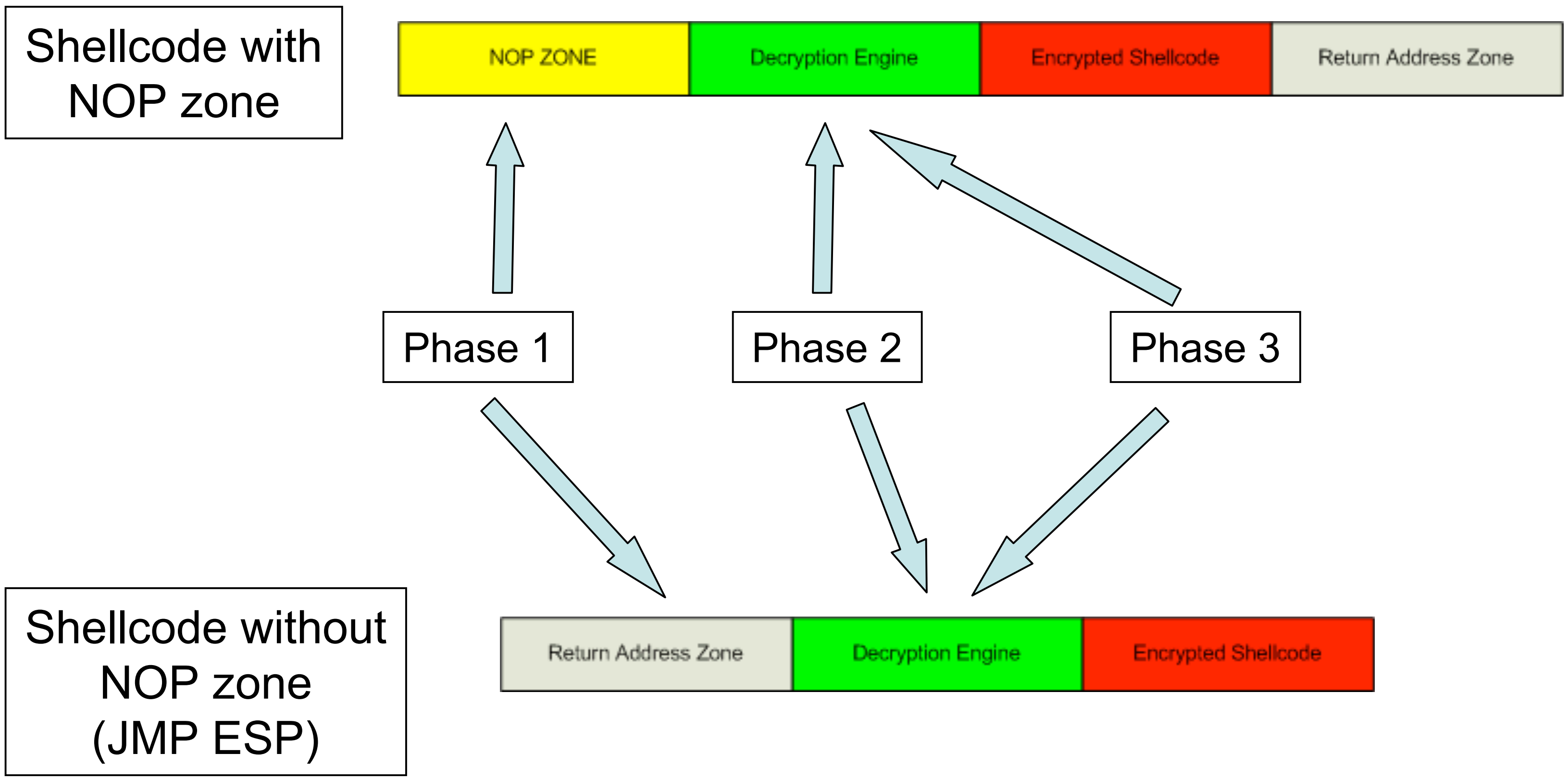

```
00002A76: 54          push  esp
00002A77: 2404        and   al,004
00002A79: 33C0        xor   eax,eax
00002A7B: 8A0A        mov   cl,[edx]
00002A7D: 84C9        test  cl,cl
00002A7F: 740F        je    .000002A90
00002A81: 80E930      sub   cl,030  ;"0"
00002A84: 8D0480      lea eax,[eax][eax]*4
00002A87: 0FB6C9      movzx ecx,cl
00002A8A: 42          inc   edx
00002A8B: 8D0441      lea   eax,[ecx][eax]*2
00002A8E: EBEB        jmps  .000002A7B
00002A90: C20400      retn  00004
```

- instructions can be found in libraries known to be always at the same memory address
- database for such instructions
- Metasploit

Detection Engine

- *Phase 1*: NOP Zone detection
  - Trigger for Phase 2
  - Can be adapted to recognize JMP ESP techniques
- *Phase 2*: Execution chain evaluation
  - Disassembling of byte stream after NOP zone
  - Evaluation of control flow instructions
- *Phase 3*: Neural network classification
  - Classification of disassembled instructions
- Implemented as SNORT Plugin

Detection Engine



Phase 1: NOP Zone Detection

- Simple detection algorithm
- Searches for consecutive NOP bytes (tests with 5 and 30 NOPS)
- NOP bytes taken from ADMmutate/CLET
- Serves as trigger for Phase 2

Phase 1: NOP Zone Detection

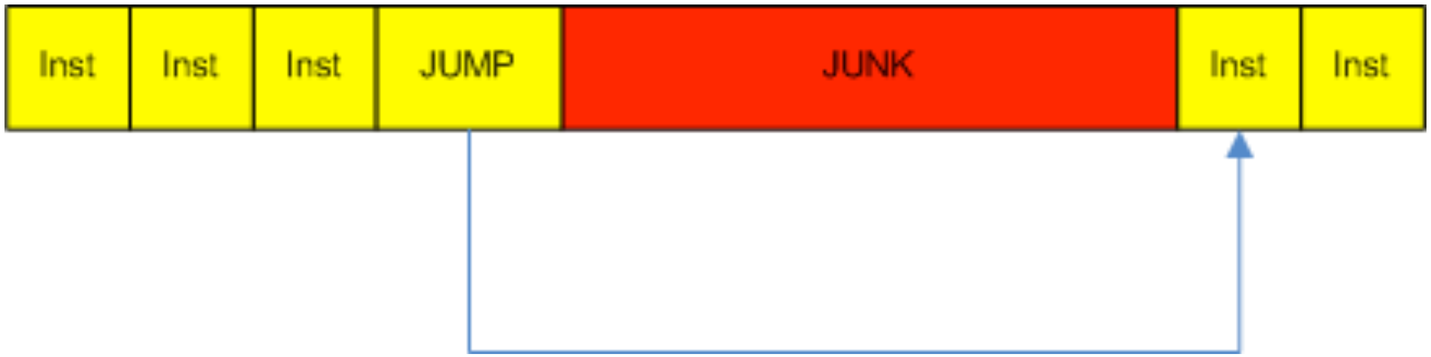
- Can be adapted to recognize shellcodes without NOP zone
- Address database for „*jmp esp*“ like instructions exist (e.g. Metasploit)
- Search for such addresses in network traffic

Phase 2: Execution Chain Evaluation

- Triggered by Phase 1
- Disassembling of bytestream after NOP zone
- Control flow instructions are evaluated
- Spectrum of instructions for each execution chain is created
- Whenever termination criterion is met NN classifies spectrum (Phase 3)

Phase 2: Execution Chain Evaluation

- Reasons:
  - decrease noise
  - parameters store encryption keys (random)
  - get instructions used by decryption engines
  - ignore junk bytes



Phase 2: Execution Chain Evaluation

DECRYPTION ENGINE

```
32    jmp short 0x67
34    pop eax
35    xor edx,edx
37    mov dl,0x20
44    rol ecx,0xb
47    add ecx,0xc29e092f
58    sub eax,0xfffffffe
5D    inc eax
63    jz 0x6c
65    jmp short 0x37
67    call 0x34
```

ENCRYPTED SHELLCODE

```
6c    xxxxxxx
```

EC1

```
32    jmp short 0x67
67    call 0x34
34    pop eax
35    xor edx,edx
37    mov dl,0x20
44    rol ecx,0xb
47    add ecx,0xc29e092f
58    sub eax,0xfffffffe
5D    inc eax
63    jz 0x6c
```

EC1B

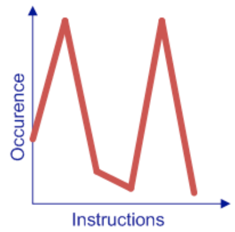
```
65    jmp short 0x37
67    mov dl,0x20
44    rol ecx,0xb
35    xor edx,edx
```

EC1A

```
6c    xxxxxxx
```

For each execution chain:

jmp: 3
xor: 2
call: 1
…



Phase 3

Phase 3: NN Classification

- *Neural network structure*:
  - 29 input neurons (29 features)
  - 12 hidden layer neurons
  - 1 output neuron
- *Training algorithm*: Levenberg-Marquardt
- *Activation function*: tansig
- *Structure* was chosen intuitively (further optimization was not necessary)

Phase 3: NN Classification

- Features are based on decryption engines of ADMmutate and CLET
- Instructions were grouped and additional instructions were added
- The last feature covers all instructions not included in the groups

Phase 3: NN classification

Feature	*Instructions*	*Feature*	*Instructions*
1	add, sub	**16**	test
2	call	**17**	shl, shr
3	and, or, not	**18**	xor
4	pop	**19**	mul, imul, fmul
5	popa	**20**	div, idiv, fdiv
6	popf	**21**	cmp, cmpsb, cmpsw…
7	push	**22**	sti, stc, std
8	pusha	**23**	neg
9	pushf	**24**	lahf
10	rol, ror	**25**	sahf
11	jcc	**26**	aaa, aad, aam, aas…
12	jmp	**27**	clc, cld, cli…
13	inc, dec	**28**	cbw, cwd, cdq, cdwe
14	loop, loope, loopne	**29**	all other instructions
15	mov		

Shellcode engines

- *ADMmutate*: XOR encryption, JUNK instructions between real decryption loop instructions
- *CLET*: XOR encryption, JUNK bytes to defeat spectrum analysis
- *JempiScodes*: XOR encryption, easy to detect
- *EE1*: XOR encryption, JUNK instructions
- *EE2*: TEA encryption, JUNK instructions
- *EE3*: Usage of different instruction for „encryption“, JUNK instructions

Results

- *Positive training data (shellcodes):*
  - About 2000 examples generated with each engine (seperated into test/train sets)
- *Negative training data:*
  - About 9 Gb of data taken from Linux/Windows installations
  - Covers executables, multimedia files, documents…

Results

- Collection of negative data:
  - Phase 1 is applied to negative test sets
  - Several million collected negative examples
  - 8000 negative examples are taken randomly
  - Initial NN is trained with those examples
  - All phases are applied to the train sets
  - Remaining examples are added to the negative training set…

	ADM	CLET	JEMPI	EE1	EE2	EE3
ADM	100%	38,8%	100%	79,2%	93%	75,9%
CLET	3,2%	100%	0%	1,7%	0%	3,5%
JEMPI	26,6%	0%	100%	13%	0,1%	17,7%
EE1	17,4%	91,2%	0,8%	100%	100%	100%
EE2	2,3%	33%	0%	4,7%	100%	1,5%
EE3	20%	98,9%	0,8%	100%	97%	100%

Phase 3: NN Classification

- Best results were taken (ADMmutate and EE3)
- New NN was trained with examples from both engines

Threshold	ADMmutate	CLET	Jempi	EE1	EE2	EE3
30	100%	***100%***	***71,4%***	***100%***	***98,3%***	100%
5	100%	***100%***	***0%***	***99,8%***	***49,3%***	100%

Analysis

- Engine can be retrained on new polymorphic shellcode engines without in depth knowledge
- Results indicate that the detection engine is capable of detecting engines not used during the training process

Outlook

- Unsupervised learning
- Use other methods to trigger Phase 2
- Automatic feature selection
- Use gained experience to implement anomaly detection system
- Intrusion detection framework: input plugins, training plugins, detection plugins based on machine learning

Thank you for your attention!