SSEXY
Binary Obfuscation using SSE

Jurriaan Bremer @skier_t

Hack in the Box Amsterdam 2012

May 24, 2012

Table of Contents

Brief Introduction

SSEXY

SSEXY Internals

Obstacles

Improvements

Last notes

Me?

Student at the University of Amsterdam

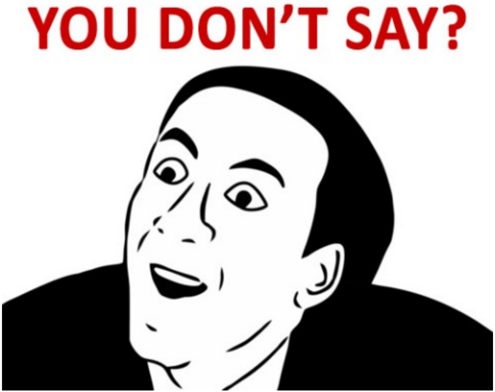
Interested in Low-Level Stuff

Hack in the Box CTF

De Eindbazen

Me?

- Student at the University of Amsterdam
- Interested in Low-Level Stuff
- Hack in the Box CTF
- De Eindbazen
- Giving this Presentation!


YOU DON'T SAY?

Portable Executables

PE File - Windows Binary

Bunch of Headers

Defines:

- Data (e.g. strings, "hello world")
- Imported Functions (e.g. **printf**)
- Metadata (e.g. Relocations)
- Code

Portable Executables

PE File - Windows Binary

Bunch of Headers

Defines:

- Data (e.g. strings, "hello world")
- Imported Functions (e.g. **printf**)
- Metadata (e.g. Relocations)
- Code

Portable Executables

PE File - Windows Binary

Bunch of Headers

Defines:

- Data (e.g. strings, "hello world")
- Imported Functions (e.g. **printf**)
- Metadata (e.g. Relocations)
- Code

Portable Executables

PE File - Windows Binary

Bunch of Headers

Defines:

- Data (e.g. strings, "hello world")
- Imported Functions (e.g. **printf**)
- Metadata (e.g. Relocations)
- Code

Portable Executables

PE File - Windows Binary

Bunch of Headers

Defines:

- Data (e.g. strings, "hello world")
- Imported Functions (e.g. **printf**)
- Metadata (e.g. Relocations)
- Code

Portable Executables

PE File - Windows Binary

Bunch of Headers

Defines:

- Data (e.g. strings, "hello world")
- Imported Functions (e.g. **printf**)
- Metadata (e.g. Relocations)
- Code

x86 Instruction Set

Variable Instruction Size

Eight 32bit General Purpose Registers

Approximately 100 "normal" Instructions

x86 Instruction Set

- Variable Instruction Size
- Eight 32bit General Purpose Registers
- Approximately 100 "normal" Instructions
- SIMD (Single Instruction, Multiple Data)
- SSE and SSE2 (Streaming SIMD Extensions)
- Pentium 3 (SSE), Pentium 4 (SSE2)
- Eight 128bit XMM Registers
- Few dozen Instructions

Traditional Virtual Machines (1)

Custom Bytecode, Custom Context

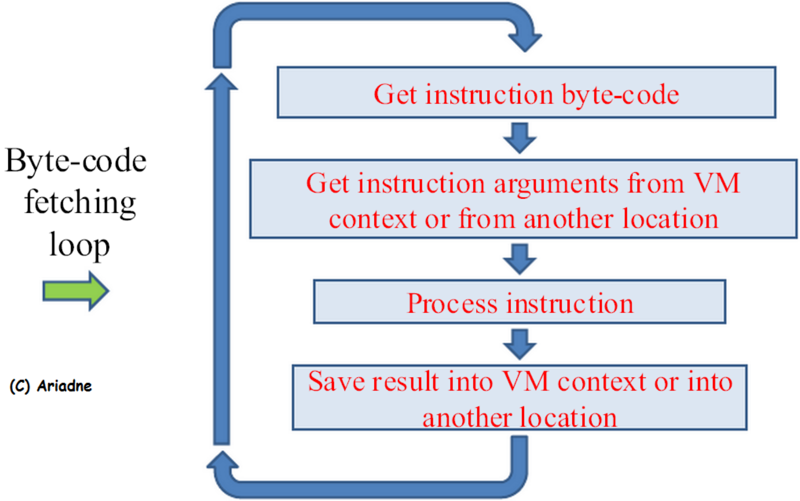
Main Execution Loop

Traditional Virtual Machines (1)

Custom Bytecode, Custom Context

Main Execution Loop

Traditional Virtual Machines (2)



Metamorphic Code

Listing 1: Before

```
mov eax, 0x100
mov ebx, 0x200
```

Listing 2: After

```
push 0x2100
pop eax
sub eax, 0x2000

mov ebx, eax
xor ebx, 0x300
```

Why SSE? (1)

Uncommon Instructions

Why SSE? (1)

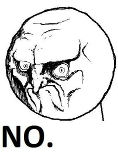
Uncommon Instructions

Did you ever encounter SSE during Reverse Engineering?

Why SSE? (1)

Uncommon Instructions

Did you ever encounter SSE during Reverse Engineering?



Why SSE? (2)

Obscure Instruction Names

Why SSE? (2)

Obscure Instruction Names

What do you think 'CVTTPD2DQ' does?

Why SSE? (2)

Obscure Instruction Names

What do you think 'CVTTPD2DQ' does?

Convert with Truncation Packed Double-Precision FP Values to Packed Dword Integers

Why SSE? (3)

Enormous Registers

Why SSE? (3)

Enormous Registers

How many General Purpose Registers fit in an XMM Register?
(This is not a trick question)

Why SSE? (3)

Enormous Registers

How many General Purpose Registers fit in an XMM Register?
(This is not a trick question)

Four..

Global SSEXY Workings

Load a PE File as Input (*pefile*)

- Parse the PE Headers
- Disassemble all Instructions (*distorm3*, !)
- Analyze Metadata (!)
- Needs Relocations
- At the moment not very generic :(

Generate a new Binary

- Translate all Instructions (*pyasm2* + *ssexy*, !)
- Craft a PE Binary (*gcc*)

Global SSEXY Workings

- Load a PE File as Input (*pefile*)
  - Parse the PE Headers
  - Disassemble all Instructions (*distorm3*, !)
  - Analyze Metadata (!)
  - Needs Relocations
  - At the moment not very generic :(
- Generate a new Binary
  - Translate all Instructions (*pyasm2* + *ssexy*, !)
  - Craft a PE Binary (*gcc*)

Global SSEXY Workings

- Load a PE File as Input (*pefile*)
  - Parse the PE Headers
  - Disassemble all Instructions (*distorm3*, !)
  - Analyze Metadata (!)
  - Needs Relocations
  - At the moment not very generic :(
- Generate a new Binary
  - Translate all Instructions (*pyasm2* + *ssexy*, !)
  - Craft a PE Binary (*gcc*)

Global SSEXY Workings

Load a PE File as Input (*pefile*)

- Parse the PE Headers
- Disassemble all Instructions (*distorm3*, !)
- Analyze Metadata (!)
- Needs Relocations
- At the moment not very generic :(

Generate a new Binary

- Translate all Instructions (*pyasm2* + *ssexy*, !)
- Craft a PE Binary (*gcc*)

Global SSEXY Workings

Load a PE File as Input (*pefile*)

- Parse the PE Headers
- Disassemble all Instructions (*distorm3*, !)
- Analyze Metadata (!)
- Needs Relocations
- At the moment not very generic :(

Generate a new Binary

- Translate all Instructions (*pyasm2* + *ssexy*, !)
- Craft a PE Binary (*gcc*)

Global SSEXY Workings

Load a PE File as Input (*pefile*)

- Parse the PE Headers
- Disassemble all Instructions (*distorm3*, !)
- Analyze Metadata (!)
- Needs Relocations
- At the moment not very generic :(

Generate a new Binary

- Translate all Instructions (*pyasm2* + *ssexy*, !)
- Craft a PE Binary (*gcc*)

Global SSEXY Workings

Load a PE File as Input (*pefile*)
- Parse the PE Headers
- Disassemble all Instructions (*distorm3*, !)
- Analyze Metadata (!)
- Needs Relocations
- At the moment not very generic :(

Generate a new Binary
- Translate all Instructions (*pyasm2* + *ssexy*, !)
- Craft a PE Binary (*gcc*)

Global SSEXY Workings

- Load a PE File as Input (*pefile*)
  - Parse the PE Headers
  - Disassemble all Instructions (*distorm3*, !)
  - Analyze Metadata (!)
  - Needs Relocations
  - At the moment not very generic :(
- Generate a new Binary
  - Translate all Instructions (*pyasm2* + *ssexy*, !)
  - Craft a PE Binary (*gcc*)

Global SSEXY Workings

- Load a PE File as Input (*pefile*)
  - Parse the PE Headers
  - Disassemble all Instructions (*distorm3*, !)
  - Analyze Metadata (!)
  - Needs Relocations
  - At the moment not very generic :(
- Generate a new Binary
  - Translate all Instructions (*pyasm2* + *ssexy*, !)
  - Craft a PE Binary (*gcc*)

pyasm2

Intel syntax:
mov ecx, dword [ebp+8]

```
>>> from pyasm2 import *
>>> mov(ecx, dword[ebp+8])
<pyasm2.mov instance at 0x....>

>>> str(mov(ecx, dword[ebp+8]))
'mov ecx, dword [ebp+0x8]'

>>> mov(ecx, dword[ebp+8]).encode()
'\x8bM\x08'
```

Theoretical

General Purpose Register stored in XMM Registers:

- Eight 32bit GPRs fit in 2 XMM Registers
- eax, ecx, edx, ebx in xmm6
- esp, ebp, esi, edi in xmm7
- xmm0, ..., xmm3 for intermediate values

General Purpose Instructions:

- Written as a sequence of SSE Instructions;
- Source operand(s) are loaded
- Some operation is performed
- Destination operand(s) are written

Theoretical

General Purpose Register stored in XMM Registers:

- Eight 32bit GPRs fit in 2 XMM Registers
- eax, ecx, edx, ebx in xmm6
- esp, ebp, esi, edi in xmm7
- xmm0, ..., xmm3 for intermediate values

General Purpose Instructions:

- Written as a sequence of SSE Instructions;
- Source operand(s) are loaded
- Some operation is performed
- Destination operand(s) are written

Theoretical

General Purpose Register stored in XMM Registers:

- Eight 32bit GPRs fit in 2 XMM Registers
- eax, ecx, edx, ebx in xmm6
- esp, ebp, esi, edi in xmm7
- xmm0, ..., xmm3 for intermediate values

General Purpose Instructions:

- Written as a sequence of SSE Instructions;
- Source operand(s) are loaded
- Some operation is performed
- Destination operand(s) are written

Theoretical

General Purpose Register stored in XMM Registers:

- Eight 32bit GPRs fit in 2 XMM Registers
- eax, ecx, edx, ebx in xmm6
- esp, ebp, esi, edi in xmm7
- xmm0, ..., xmm3 for intermediate values

General Purpose Instructions:

- Written as a sequence of SSE Instructions;
- Source operand(s) are loaded
- Some operation is performed
- Destination operand(s) are written

Theoretical

General Purpose Register stored in XMM Registers:

- Eight 32bit GPRs fit in 2 XMM Registers
- eax, ecx, edx, ebx in xmm6
- esp, ebp, esi, edi in xmm7
- xmm0, ..., xmm3 for intermediate values

General Purpose Instructions:

- Written as a sequence of SSE Instructions;
- Source operand(s) are loaded
- Some operation is performed
- Destination operand(s) are written

Theoretical

General Purpose Register stored in XMM Registers:

- Eight 32bit GPRs fit in 2 XMM Registers
- eax, ecx, edx, ebx in xmm6
- esp, ebp, esi, edi in xmm7
- xmm0, ..., xmm3 for intermediate values

General Purpose Instructions:

- Written as a sequence of SSE Instructions;
- Source operand(s) are loaded
- Some operation is performed
- Destination operand(s) are written

Theoretical

General Purpose Register stored in XMM Registers:

- Eight 32bit GPRs fit in 2 XMM Registers
- eax, ecx, edx, ebx in xmm6
- esp, ebp, esi, edi in xmm7
- xmm0, ..., xmm3 for intermediate values

General Purpose Instructions:

- Written as a sequence of SSE Instructions;
- Source operand(s) are loaded
- Some operation is performed
- Destination operand(s) are written

Theoretical

General Purpose Register stored in XMM Registers:

- Eight 32bit GPRs fit in 2 XMM Registers
- eax, ecx, edx, ebx in xmm6
- esp, ebp, esi, edi in xmm7
- xmm0, ..., xmm3 for intermediate values

General Purpose Instructions:

- Written as a sequence of SSE Instructions;
- Source operand(s) are loaded
- Some operation is performed
- Destination operand(s) are written

Practical

Listing 3: Before

```
mov ecx, [ebp+8]
```

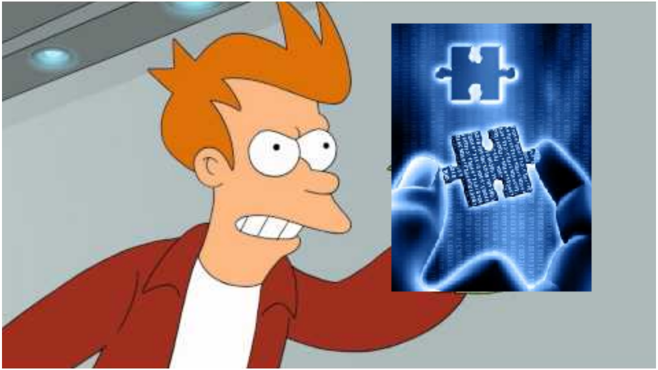
Listing 4: After

```
movd xmm3, [__m32_0]
pshufd xmm2, xmm7, 1
paddd xmm3, xmm2

movd eax, xmm3
mov eax, [eax]
movd xmm0, eax

pshufd xmm0, xmm0, 0
pand xmm0, [__m128_0]
pand xmm6, [__m128_1]
pxor xmm6, xmm0
```

Shut up and take my Binary! (1)



Shut up and take my Binary! (2)

Listening daemon which checks the input against a hardcoded hash and executes it when the hash matches.

```
unsigned short hash(const unsigned char *s,
        unsigned int len)
{
    unsigned int ret = 0;
    while (len--) ret += *s++ * len;
    return ret;
}
```

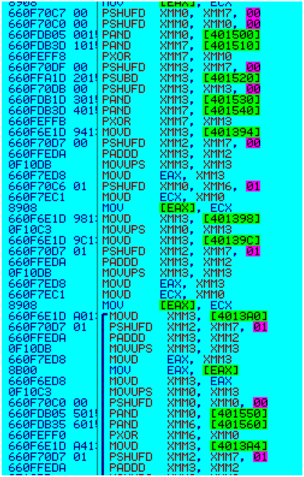
Shut up and take my Binary! (3)

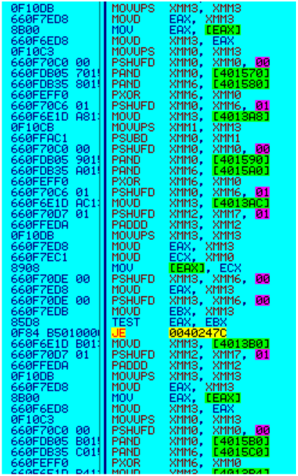
```
660F70C7 00   PSHUFD  XMM0, XMM7, 00
660F70C0 00   PSHUFD  XMM0, XMM0, 00
660FDB05 001! PAND    XMM0, [401500]
660FDB3D 101! PAND    XMM7, [401510]
660FEFF8      PXOR    XMM7, XMM0
660F70DF 00   PSHUFD  XMM3, XMM7, 00
660FFA1D 201! PSUBD   XMM3, [401520]
660F70DB 00   PSHUFD  XMM3, XMM3, 00
660FDB1D 301! PAND    XMM3, [401530]
660FDB3D 401! PAND    XMM7, [401540]
660FEFFB      PXOR    XMM7, XMM3
660F6E1D 941: MOVD    XMM3, [401394]
660F70D7 00   PSHUFD  XMM2, XMM7, 00
660FFEDA      PADDD   XMM3, XMM2
0F10DB        MOVUPS  XMM3, XMM3
660F7ED8      MOVD    EAX, XMM3
660F70C6 01   PSHUFD  XMM0, XMM6, 01
660F7EC1      MOVD    ECX, XMM0
8908          MOV     [EAX], ECX
660F6E1D 981: MOVD    XMM3, [401398]
0F10C3        MOVUPS  XMM0, XMM3
660F6E1D 9C1: MOVD    XMM3, [40139C]
660F70D7 01   PSHUFD  XMM2, XMM7, 01
660FFEDA      PADDD   XMM3, XMM2
0F10DB        MOVUPS  XMM3, XMM3
660F7ED8      MOVD    EAX, XMM3
660F7EC1      MOVD    ECX, XMM0
8908          MOV     [EAX], ECX
660F6E1D A01: MOVD    XMM3, [4013A0]
660F70D7 01   PSHUFD  XMM2, XMM7, 01
660FFEDA      PADDD   XMM3, XMM2
0F10DB        MOVUPS  XMM3, XMM3
660F7ED8      MOVD    EAX, XMM3
8B00          MOV     EAX, [EAX]
660F6ED8      MOVD    XMM3, EAX
0F10C3        MOVUPS  XMM0, XMM3
660F70C0 00   PSHUFD  XMM0, XMM0, 00
660FDB05 501! PAND    XMM0, [401550]
660FDB35 601! PAND    XMM6, [401560]
660FEFF0      PXOR    XMM6, XMM0
660F6E1D A41: MOVD    XMM3, [4013A4]
660F70D7 01   PSHUFD  XMM2, XMM7, 01
660FFEDA      PADDD   XMM3, XMM2
```

```
0F10DB        MOVUPS  XMM3, XMM3
660F7ED8      MOVD    EAX, XMM3
8B00          MOV     EAX, [EAX]
660F6ED8      MOVD    XMM3, EAX
0F10C3        MOVUPS  XMM0, XMM3
660F70C0 00   PSHUFD  XMM0, XMM0, 00
660FDB05 701! PAND    XMM0, [401570]
660FDB35 801! PAND    XMM6, [401580]
660FEFF0      PXOR    XMM6, XMM0
660F70C6 01   PSHUFD  XMM0, XMM6, 01
660F6E1D A81: MOVD    XMM3, [4013A8]
0F10CB        MOVUPS  XMM1, XMM3
660FFAC1      PSUBD   XMM0, XMM1
660F70C0 00   PSHUFD  XMM0, XMM0, 00
660FDB05 901! PAND    XMM0, [401590]
660FDB35 A01! PAND    XMM6, [4015A0]
660FEFF0      PXOR    XMM6, XMM0
660F70C6 01   PSHUFD  XMM0, XMM6, 01
660F6E1D AC1: MOVD    XMM3, [4013AC]
660F70D7 01   PSHUFD  XMM2, XMM7, 01
660FFEDA      PADDD   XMM3, XMM2
0F10DB        MOVUPS  XMM3, XMM3
660F7ED8      MOVD    EAX, XMM3
660F7EC1      MOVD    ECX, XMM0
8908          MOV     [EAX], ECX
660F70DE 00   PSHUFD  XMM3, XMM6, 00
660F7ED8      MOVD    EAX, XMM3
660F70DE 00   PSHUFD  XMM3, XMM6, 00
660F7EDB      MOVD    EBX, XMM3
85D8          TEST    EAX, EBX
0F84 B501000( JE      0040247C
660F6E1D B01: MOVD    XMM3, [4013B0]
660F70D7 01   PSHUFD  XMM2, XMM7, 01
660FFEDA      PADDD   XMM3, XMM2
0F10DB        MOVUPS  XMM3, XMM3
660F7ED8      MOVD    EAX, XMM3
8B00          MOV     EAX, [EAX]
660F6ED8      MOVD    XMM3, EAX
0F10C3        MOVUPS  XMM0, XMM3
660F70C0 00   PSHUFD  XMM0, XMM0, 00
660FDB05 B01! PAND    XMM0, [4015B0]
660FDB35 C01! PAND    XMM6, [4015C0]
660FEFF0      PXOR    XMM6, XMM0
```

3 lines of C

Shut up and take my Binary! (3)

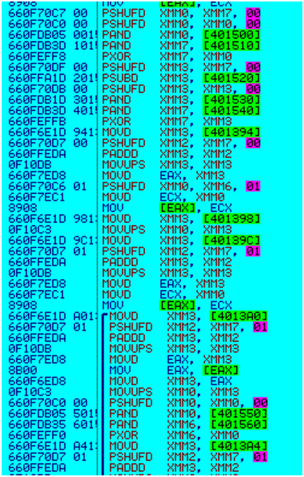


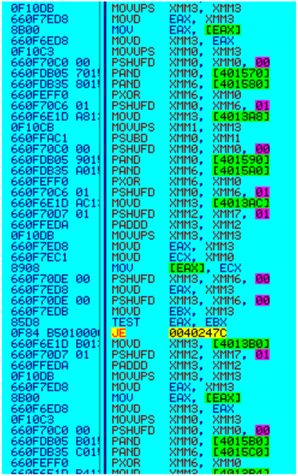


3 lines of C

20 lines of x86

Shut up and take my Binary! (3)





3 lines of C

20 lines of x86

200 lines of SSE

Shut up and take my Binary! (4)

Demonstration

What I will demonstrate:

- Start ssexified listening daemon
- Send a few invalid and valid packets to the daemon (netcat)
- Show that only the valid ones are executed

Obstacles - Memory Addresses (1)

x86 supports memory addresses

'effective address', 'scale index' and 'displacement'

For example: dword ptr [eax+ebx*4+32]

effective address: *eax*

scale index: *ebx*, multiplied by *four*

displacement: *32*

SSEXY has to emulate this

Obstacles - Memory Addresses (1)

x86 supports memory addresses

'effective address', 'scale index' and 'displacement'

For example: dword ptr [eax+ebx*4+32]

effective address: *eax*

scale index: *ebx*, multiplied by *four*

displacement: *32*

SSEXY has to emulate this

Obstacles - Memory Addresses (1)

x86 supports memory addresses

'effective address', 'scale index' and 'displacement'

For example: dword ptr [eax+ebx*4+32]

effective address: *eax*

scale index: *ebx*, multiplied by *four*

displacement: *32*

SSEXY has to emulate this

Obstacles - Memory Addresses (1)

x86 supports memory addresses

'effective address', 'scale index' and 'displacement'

For example: dword ptr [eax+ebx*4+32]

effective address: *eax*

scale index: *ebx*, multiplied by *four*

displacement: *32*

SSEXY has to emulate this

Obstacles - Memory Addresses (1)

x86 supports memory addresses

'effective address', 'scale index' and 'displacement'

For example: dword ptr [eax+ebx*4+32]

effective address: *eax*

scale index: *ebx*, multiplied by *four*

displacement: *32*

SSEXY has to emulate this

Obstacles - Memory Addresses (1)

x86 supports memory addresses

'effective address', 'scale index' and 'displacement'

For example: dword ptr [eax+ebx*4+32]

effective address: *eax*

scale index: *ebx*, multiplied by *four*

displacement: *32*

SSEXY has to emulate this

Obstacles - Memory Addresses (1)

x86 supports memory addresses

'effective address', 'scale index' and 'displacement'

For example: dword ptr [eax+ebx*4+32]

effective address: *eax*

scale index: *ebx*, multiplied by *four*

displacement: *32*

SSEXY has to emulate this

Obstacles - Memory Addresses (2)

Remember the 'Practical' slide?

```
movd xmm3, [__m32_0]    ; load the displacement
pshufd xmm2, xmm6, 0    ; load effective address
paddd xmm3, xmm2
pshufd xmm2, xmm6, 3    ; load the scale index
pslld xmm2, 2           ; multiply by four
paddd xmm3, xmm2
movd eax, xmm3          ; store address to a gpr

mov ecx, dword [eax]    ; read from address
```

Memory address is now in *xmm3* and *eax*

Using *eax* we can read/write from/to this address

Obstacles - Memory Addresses (2)

Remember the 'Practical' slide?

```
movd xmm3, [__m32_0]     ; load the displacement
pshufd xmm2, xmm6, 0     ; load effective address
paddd xmm3, xmm2
pshufd xmm2, xmm6, 3     ; load the scale index
pslld xmm2, 2            ; multiply by four
paddd xmm3, xmm2
movd eax, xmm3           ; store address to a gpr

mov ecx, dword [eax]     ; read from address
```

Memory address is now in *xmm3* and *eax*

Using *eax* we can read/write from/to this address

Obstacles - Memory Addresses (2)

Remember the 'Practical' slide?

```
movd xmm3, [__m32_0]    ; load the displacement
pshufd xmm2, xmm6, 0    ; load effective address
paddd xmm3, xmm2
pshufd xmm2, xmm6, 3    ; load the scale index
pslld xmm2, 2           ; multiply by four
paddd xmm3, xmm2
movd eax, xmm3          ; store address to a gpr

mov ecx, dword [eax]    ; read from address
```

Memory address is now in *xmm3* and *eax*

Using *eax* we can read/write from/to this address

Obstacles - Conditional Jumps

No usable comparison instructions

Use normal x86 instructions instead

Obstacles - Conditional Jumps

No usable comparison instructions

Use normal x86 instructions instead

Obstacles - Function Calls

Two types; *internal* and *external*

internal: function calls within the application

external: function calls to 3rd party libraries (e.g. windows API)

internal: these are generally fine.. :)

external: need extra precautions;

stack pointer has to be set

Funky stuff happens, e.g. *MessageBoxA* resets xmm registers

Obstacles - Function Calls

Two types; *internal* and *external*

internal: function calls within the application

external: function calls to 3rd party libraries (e.g. windows API)

internal: these are generally fine.. :)

external: need extra precautions;

stack pointer has to be set

Funky stuff happens, e.g. *MessageBoxA* resets xmm registers

Obstacles - Function Calls

Two types; *internal* and *external*

internal: function calls within the application

external: function calls to 3rd party libraries (e.g. windows API)

internal: these are generally fine.. :)

external: need extra precautions;

stack pointer has to be set

Funky stuff happens, e.g. *MessageBoxA* resets xmm registers

Obstacles - Function Calls

Two types; *internal* and *external*

internal: function calls within the application

external: function calls to 3rd party libraries (e.g. windows API)

internal: these are generally fine.. :)

external: need extra precautions;

stack pointer has to be set

Funky stuff happens, e.g. *MessageBoxA* resets xmm registers

Obstacles - Function Calls

Two types; *internal* and *external*

internal: function calls within the application

external: function calls to 3rd party libraries (e.g. windows API)

internal: these are generally fine.. :)

external: need extra precautions;

stack pointer has to be set

Funky stuff happens, e.g. *MessageBoxA* resets xmm registers

Obstacles - Function Calls

Two types; *internal* and *external*

internal: function calls within the application

external: function calls to 3rd party libraries (e.g. windows API)

internal: these are generally fine.. :)

external: need extra precautions;

stack pointer has to be set

Funky stuff happens, e.g. *MessageBoxA* resets xmm registers

Obstacles - Function Calls

Two types; *internal* and *external*

internal: function calls within the application

external: function calls to 3rd party libraries (e.g. windows API)

internal: these are generally fine.. :)

external: need extra precautions;

stack pointer has to be set

Funky stuff happens, e.g. *MessageBoxA* resets xmm registers

Shuffling stored GPRs

GPRs	eax	ecx	edx	ebx
Before	a0 a1 a2 a3	c0 c1 c2 c3	d0 d1 d2 d3	b0 b1 b2 b3
After	a0 c0 d0 b0	a1 c1 d1 b1	a2 c2 d2 b2	a3 c3 d3 b3

The values of GPRs are much harder to read

Shuffling stored GPRs

GPRs	eax	ecx	edx	ebx
Before	a0 a1 a2 a3	c0 c1 c2 c3	d0 d1 d2 d3	b0 b1 b2 b3
After	a0 c0 d0 b0	a1 c1 d1 b1	a2 c2 d2 b2	a3 c3 d3 b3

The values of GPRs are much harder to read

Every read/write operation to GPRs need (de-)shuffling code

Shuffling stored GPRs

GPRs	eax	ecx	edx	ebx
Before	a0 a1 a2 a3	c0 c1 c2 c3	d0 d1 d2 d3	b0 b1 b2 b3
After	a0 c0 d0 b0	a1 c1 d1 b1	a2 c2 d2 b2	a3 c3 d3 b3

The values of GPRs are much harder to read

Every read/write operation to GPRs need (de-)shuffling code

Possibly use different encodings per function

(Needs translation when calling another function)

Encrypting stored GPRs

Each GPR is encrypted, e.g. using a unique xor key

Encrypting stored GPRs

Each GPR is encrypted, e.g. using a unique xor key

Every read/write to GPRs need decryption/encryption code

Encrypting stored GPRs

Each GPR is encrypted, e.g. using a unique xor key

Every read/write to GPRs need decryption/encryption code

Again, can be function-specific

Getting rid of the x86 mov instruction (1)

Did you notice the *mov* instruction earlier? (in the Obstacles - Memory Addresses slide)

mov is an x86 general purpose instruction

SSEXY doesn't like those

Getting rid of the x86 mov instruction (1)

Did you notice the *mov* instruction earlier? (in the Obstacles - Memory Addresses slide)

mov is an x86 general purpose instruction

SSEXY doesn't like those

Getting rid of the x86 mov instruction (1)

Did you notice the *mov* instruction earlier? (in the Obstacles - Memory Addresses slide)

mov is an x86 general purpose instruction

SSEXY doesn't like those

Getting rid of the x86 mov instruction (2)

To refresh your memory

```
movd xmm3, [__m32_0]     ; load the displacement
pshufd xmm2, xmm6, 0     ; load effective address
paddd xmm3, xmm2
pshufd xmm2, xmm6, 3     ; load the scale index
pslld xmm2, 2            ; multiply by four
paddd xmm3, xmm2
movd eax, xmm3           ; store address to a gpr

mov ecx, dword [eax]     ; read from address
```

Getting rid of the x86 mov instruction (3)

```
movd eax, xmm3          ; still refreshing
mov ecx, dword [eax]    ; your memory ;)
```

movd takes the lowest 32bits of *xmm3* and stores them into *eax*

Now if we rewrite the *mov* instruction

Getting rid of the x86 mov instruction (3)

```
movd eax, xmm3            ; still refreshing
mov ecx, dword [eax]      ; your memory ;)
```

movd takes the lowest 32bits of *xmm3* and stores them into *eax*

Now if we rewrite the *mov* instruction

```
movd dword [next_instr+4], xmm3
next_instr:
movd xmm2, dword [0x11223344]
```

Getting rid of the x86 mov instruction (3)

```
movd eax, xmm3            ; still refreshing
mov ecx, dword [eax]      ; your memory ;)
```

movd takes the lowest 32bits of *xmm3* and stores them into *eax*
Now if we rewrite the *mov* instruction

```
movd dword [next_instr+4], xmm3
next_instr:
movd xmm2, dword [0x11223344]
```

Single-threaded only (we overwrite machine code)

Getting rid of the x86 mov instruction (3)

```
movd eax, xmm3              ; still refreshing
mov ecx, dword [eax]        ; your memory ;)
```

movd takes the lowest 32bits of *xmm3* and stores them into *eax*

Now if we rewrite the *mov* instruction

```
movd dword [next_instr+4], xmm3
next_instr:
movd xmm2, dword [0x11223344]
```

Single-threaded only (we overwrite machine code)

Doesn't matter, had SSE.

Performance

Running the hash() algorithm a few million times, showed me a performance decrease of 5 times.
Sounds reasonable, since it takes a Reverse Engineer probably five times longer to analyze the binary.. ;)

Source?-s

http://jbremer.org/

http://github.com/jbremer/ssexy

jurriaanbremer@gmail.com